Artificial Software Diversification for WebAssembly

JAVIER CABRERA-ARTEAGA

Licentiate Thesis in [Research Subject - as it is in your ISP]
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden [2022]

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT XXXX:XX
ISBN XXX-XX-XXXX-XXX-X

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av licentiatexamen i [ämne/subject] [veckodag/weekday] den [dag/day] [månad/month] [år/2022] klockan [tid/time] i [sal/hall], Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

WebAssembly has become the fourth official web language. This new language allows web browsers to execute existing programs or libraries written in other languages, such as C/C++ and Rust. Apart from web browsers, WebAssembly evolves to be part of Edge-Cloud computing platforms. Despite being designed with security as a premise, it is not exempt from vulnerabilities. Our approaches deal with this fact by providing a preemptive solution with software diversification.

In this thesis, we propose an automatic approach to generate software diversification for WebAssembly programs. In addition, we provide complementary implementation for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for WebAssembly. Our results show that our approaches can provide an automated end-to-end solution for the diversification of WebAssembly programs. The main contributions of this work are:

- We highlight the lack of diversification techniques for WebAssembly through an exhaustive literature review.
- We provide the implementation of two tools, CROW and MEWE. These tools provide randomization and multivariant execution for respectively.
- We include *constant inferring* as a new code transformation to generate software diversification for WebAssembly.
- We empirically demonstrate the impact of our technique by evaluating the static and dynamic behavior of the generated diversification.

Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time. Therefore, our approaches harden unknown and yet-unknown vulnerabilities.

Keywords: WebAssembly, Software Diversification, Automatic Software Engineering, Security

Sammanfattning

Write your Swedish summary (popular description) here...
Keywords: Keyword1, keyword2, ...

Acknowledgements

Paraphrasing a good friend of mine: the persons that contributed to this work know who they are, and I prefer to thank them personally.

Javier Cabrera-Arteaga,
Stockholm, May 2022

Contents

Part I

Thesis

01 INTRODUCTION

"Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a démouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations. "

— MR. Cox ***1893**, Cinderella: Three hundred and forty-five variants [**?**]*

The Web Consortium (W3C) standarized bytecode for the web environment with the WebAssembly (Wasm) language in 2015. Wasm allows web browsers to execute existing programs or libraries written in other languages, such as C/C++ and Rust. Beyond web environments, WebAssembly evolves to be part of Edge-Cloud computing platforms [**?** **?**]. Despite being designed for sandboxing and secure execution, it is not exempt from vulnerabilities [**?**]. For example, WebAssembly engines are vulnerable to speculative execution [**?**], and C/C++ source code vulnerabilities might be ported to Wasm binaries [**?**].

One strategy to hide such vulnerabilities is to move them in time as a preemptive solution. The goal is to make potential vulnerabilities available only in a time window. This makes potential attackers not hit what they cannot see. This strategy is usually called Moving Target Defense (MTD) [**?** **?**]. MTD for software is a collection of techniques that aim to improve the security of a system by constantly rotating its vulnerable programs from one variant to another. A program variant should be different from the original program but functionally equivalent to it. By rotating the deployment and execution between the program variants, a potential attacker needs more efforts to perform the same attack for all variants [**?**]. Thus, one premise for effectively implementing MTD for a given program is the need for the program variants.

In MTD, Software Diversification is the process of finding, creating, and deploying program variants. Usually, program variants could be found in the wild in a phenomenon called natural diversity [**?**]. In the case of WebAssembly, since it is a novel technology, there is no natural diversity. Thus, effective MTD cannot be implemented due to the lack of program variants. This work proposes to create program variants for WebAssembly artificially. Therefore, we aim to generate artificial software diversification for WebAssembly. To reach such a goal, we answer three research questions enunciated in the following.

1.1 Research questions

In this section, we present our three research questions. Our research questions are formulated by merging our publications and experiences during the creation of Software Diversification for WebAssembly.

RQ_1 **To what extent can we artifically generate program variants for WebAssembly?**
With this research question, we quantitatively assess the static differences between program variants created by our approach. We answer this question at the population level, where a program population is the collection of one original program and its generated variants. We aim to investigate the code properties that increases(or diminishes) generated diversification at population level.

RQ_2 **To what extent are the generated variants dynamically different?**
With this research question, we complement RQ_1. We aim to investigate the impact on execution traces and execution times of the generated program variants.

RQ_3 **To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?**
With this research question, we aim to investigate the impact of Software Diversification for WebAssembly in an emerging technology, Edge-Cloud computing. We evaluate the impact of a novel multivariant execution approach on real-world WebAssembly programs in a world-wide scale experiment.

1.2 Contributions

This thesis contributes through four milestones. First, as a theoretical contribution, we summarize the code transformations used to artifically generate software diversification through an exhaustive literature review. Consequently, we highlight the lack of diversification techniques for WebAssembly. Second, as a technical contribution, we provide two tools, CROW [**?**] and MEWE [**?**]. Besides, we summarize the main challenges faced during their implementation. In addition, we discuss the incorporation of *constant inferring* as a new transformation. Third, we propose a methodology to quantitatively evaluate the impact of our tools, assessing the creation of artificial software diversification for WebAssembly. Fourth and final, we empirically demonstrate the impact of our technique by evaluating the static and dynamic behavior of the generated diversification.

1.3 Publications

This work is based on the following publications:

P_1 Superoptimization of WebAssembly Bytecode [**?**]
Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Programming 2020, MoreVMs'20

P_2 CROW: Code Diversification for WebAssembly [**?**]
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
NDSS 2021, MADWeb

P_3 Multi-Variant Execution at the Edge [**?**]
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Under review

P_4 Scalable Comparison of JavaScript V8 Bytecode Traces [**?**]
Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
SPLASH 2019, VMIL

Thesis layout

This dissertation is organized in five chapters including this. Chapter 2 presents background and the state of the art for WebAssembly and Artificial Software Diversification. Chapter 3 describes our technical contributions, faced challenges and engineering decisions carried out to implement our artifacts. Chapter 4 describes the methodology followed to answer the three main research questions driving this thesis. Chapter 5 details the main results of this work. Chapter 6 concludes and discuss future work. In addition, this dissertation contains the collection of research papers previously mentioned in this chapter.

02 BACKGROUND & STATE OF THE ART

This chapter discusses the state of the art in the areas of *WebAssembly* and *Software Diversification*. In Section 2.1 we discuss the WebAssembly language, its motivation, how WebAssembly binaries are generated, the language specification, and security-related issues. In Section 2.2, we present a summary of Software Diversification, its foundational concepts and highlighted related works. We select the discussed works by their novelty, critical insights, and representativeness of their techniques. In Section 2.3, we finalize the chapter by highlighting open challenges in state-of-the-art related works.

2.1 WebAssembly overview

JavaScript is currently used in all modern web browsers to allow client-side scripting. However, due to the complexity of this language and to gain in performance and its security flaws, several alternatives appeared. For example, Java applets were introduced on web pages late in the 90's to execute Java bytecode in the client side [**?**]. Similarly, Microsoft made two attempts with ActiveX in 1996 [**?**], and with Silverlight in 2007 [**?**]. All these attempts failed to persist or had low adoption, mainly due to security issues and the lack of consensus on the community of browser vendors.

In 2014, Alon Zakai and colleagues proposed the Emscripten tool [**?**]. Emscripten used a strict subset of JavaScript, asm.js, to allow low level code such as C to be compiled to JavaScript. Asm.js was first announced as an LLVM backend [**?**]. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [**?**] compared to standard JavaScript code. The main reason why asm.js is faster, is that it limits the language features to those that can be optimized in the LLVM pipeline. Besides, it removes the majority of the dynamic characteristics of the language, limiting it to numerical types, top-level functions, and one large array in the memory directly accessed as raw data. Since asm.js is a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standarization. The work of Van Es et al. [**?**] proposed to shrink JavaScript to asm.js in a source-to-source strategy, closing the cycle and extending the fact that asm.js was mainly a compilation target for C/C++ code. Moreover, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the

parsing and the recompilation of the JavaScript code which implies a significant overhead. Consequently, the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2015. WebAssembly is a binary instruction format for a stack-based virtual machine and was officially consolidated later by the work of Haas et al. [**?**] in 2017. The announcement of WebAssembly marked the first step into the standarization of bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it outperforms asm.js [**?**]. Since 2017, the adoption of WebAssembly keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm Wasmlike is the case of a full Minecraft version [2]; and the case of Blazor [3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

■ 2.1.1 From source to Wasm

All WebAssembly programs are compiled ahead-of-time from source languages. LLVM includes Wasm as a backend since release 7.1.0 published in May 2019[4], supporting a broad range of frontend languages such as C/C++, Rust, Go or AssemblyScript[5]. The resulting binary works similarly to a traditional shared library, it includes instruction codes, symbols and exported functions. In Figure 2.1, we illustrate the workflow from the creation of Wasm binaries to their execution in the browser. The process starts by compiling the source code program to Wasm (Step ①). This step includes ahead-of-time optimizations such as optimizations in the LLVM toolchain.

The step ② builds the standard library for Wasm usually as JavaScript code. This code includes the external functions that the Wasm binary needs for its execution inside the host engine. For example, the functions to interact with the DOM of the HTML page are imported in the Wasm binary during its call from the JavaScript code. The standard library can be manually written, however, compilers like Emscripten, Rust and Binaryen can generate it automatically, making this process completely transparent to developers.

Finally, the third step (Step ③), includes the compilation and execution of the client-side code. Most of the browser engines compile either the Wasm and JavaScript codes to machine code. In the case of JavaScript, this process involves JIT and hot code replacement during runtime. For Wasm, since it is closer to machine code, and it is already optimized, this process is a one-to-one mapping. For instance, in the case of V8, the compilation process only applies simple and fast optimizations such as constant folding and dead code removal. Once V8 completes

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

[4] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`

[5] subset of the TypeScript language

the compilation process, the generated machine code for Wasm does not change anymore and is the same used along all its executions. This analysis was validated by conversations with the V8's dev team and by experimental studies in our previous contributions [?].

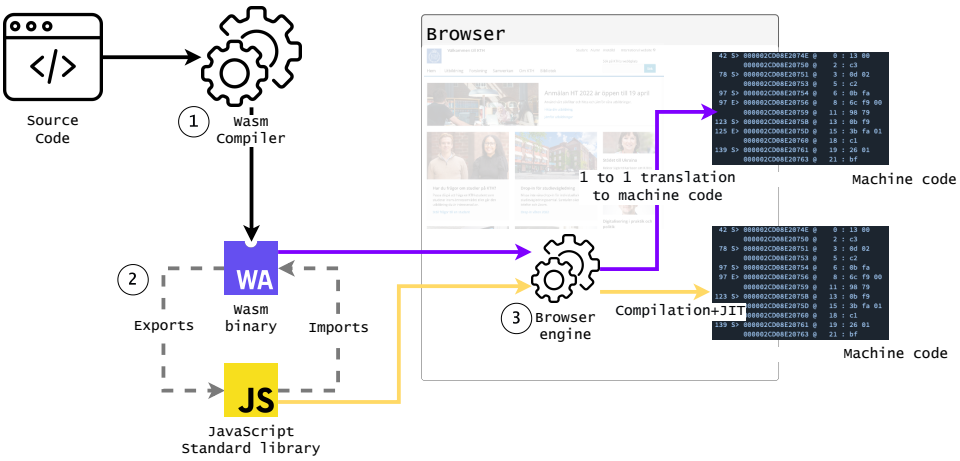


Figure 2.1: WebAssemblybuilding, compilation in the host engine and execution.

Wasm can execute directly and is platform independent. As such it is useful for IoT and Edge computing [? ?]. For instance, Cloudflare and Fastly adapted their platforms to provide Function as a Service (FaaS) directly with WebAssembly. In this case, the standard library, instead of JavaScript, is provided by any other language stack that the host environment supports. In 2019, the Bytecode Alliance [6] proposed WebAssembly System Interface (WASI) [?]. WASI is the foundation to build Wasm code outside the browser with a POSIX system interface platform. WASI standarizes the adoption of WebAssembly in heterogeneous platforms [?].

■ 2.1.2 WebAssembly specification

WebAssembly defines its own Instruction Set Architecture (ISA) [?]. It is an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, Wasm is platform independent. The ISA of Wasm includes also the necessary components that the binary requires to run in any host engine. A Wasm binary has a unique module as its main component. A module is composed by sections, corresponding to 13 types[7], each of them with an explicit semantic and a specific order inside the module. This makes the compilation to machine code faster.

In Listing 2.1 and Listing 2.2 we illustrate a C program and its compilation to Wasm. The C function contains: heap allocation, external function declaration and

[6] https://bytecodealliance.org/
[7] https://webassembly.github.io/spec/core/binary/modules.html#sections

the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.2 is in the textual format for the generated Wasm. The module in this case first defines the signature of the functions(Line 2, Line 3 and Line 4) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (Line 5, Line 35 and Line 36). The definition of the function (Line 6) and its body follows the last import declaration at Line 5.

The function body is composed by local variable declarations and typed instructions that are evaluated in a virtual stack (Line 7 to Line 32 in Listing 2.2). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at Line 32. A valid Wasm binary should have a valid stack structure that is verified during its translation to machine code. The stack validation is carried out using the static types of Wasm, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. As the listing shows, instructions are annotated with a numeric type.

Wasm manages the memory in a restricted way. A Wasm module has a linear memory component that is accessed as `i32` pointers and should be isolated from the virtual stack. The declaration of the linear data in the memory is showed in Line 37. The memory access is illustrated in Line 15. This memory is usually bound in browser engines to 2Gb of size, and it is only shareable between the process that instantiate the Wasm binary and the binary itself (explicitly declared in Line 33 and Line 36). Therefore, this ensures the isolation of the execution of Wasm code.

Wasm also provides global variables in their four primitive types. Global variables (Line 34) are only accessible by their declaration index, and it is not possible to dynamically address them. For functions, Wasm follows the same mechanism, either the functions are called by their index (Line 30) or using a static table of function declarations. This latter allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is reponsible of populating the static table of functions.

In Wasm, all instructions are grouped into blocks, being the start of a function the root block. Two consecutive block declarations can be appreciated in Line 10 and Line 11 of Listing 2.2. Control flow structures jump between block boundaries and not to any position in the code like regular assembly code. A block may specify the state that the stack must have before its execution and the result stack value coming from its instructions. Inside the Wasm binary the blocks explicitly define where they start and end (Line 25 and Line 28). By design, each block executes independently and cannot execute or refer to outer block values. This is guarantieed by explicitly annotating the state of the stack before and after the block. Three instructions handle the navigation between blocks: unconditional break, conditional break (Line 19 and Line 24) and table break. Each break instruction can only jump to one of its enclosing blocks. For example, in Listing 2.2, Line 19 forces the

Listing 2.1: Example C function.

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.2: WebAssembly code for Listing 2.1.

```
1  (module
2    (type (;0;) (func (param f32) (result i32)))
3    (type (;1;) (func))
4    (type (;2;) (func (result i32)))
5    (import "env" "ftoi" (func $ftoi (type 0)))
6    (func $main (type 2) (result i32)
7      (local i32 i32)
8      i32.const -1000
9      local.set 0
10     block  ;label = @1;
11       loop  ;label = @2;
12         i32.const 0
13         local.get 0
14         i32.add
15         i32.load
16         local.tee 1
17         i32.const 101
18         i32.ge_s
19         br_if 1 ;@1;
20         local.get 0
21         i32.const 4
22         i32.add
23         local.tee 0
24         br_if 0 ;@2;
25       end
26       i32.const 0
27       return
28     end
29     f32.const 0x1.9147aep+3
30     call $ftoi
31     local.get 1
32     i32.add)
33   (memory (;0;) 1)
34   (global (;4;) i32 (i32.const 1000))
35   (export "memory" (memory 0))
36   (export "A" (global 2))
37   (data $data (0) "\00\00\00\00...")
38 )
```

execution to jump to the end of the first block at Line 10 if the value at the top of the stack is greater than zero.

■ 2.1.3 WebAssembly security

As we described, WebAssembly is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [**?**] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

WebAssembly is vulnerable [**?**]. Implementations in both browsers and

standalone runtimes [**?**] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [**?**]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [**?**] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap using stackoverflow. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for WebAssembly. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [**?**]. Several proposals for extending WebAssembly in the current roadmap could address some existing vulnerabilities. For example, having multiple memories [8] could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors [9].

■ 2.2 Software Diversification

Software Diversification has been widely studied in the past decades. This section discusses its state of the art. Software diversification consists in synthesizing, reusing, distributing, and executing different, functionally equivalent programs. According to the survey by Baudry et al. [**?**], the motivation for software diversification can be separated in five categories: reusability [**?**], software testing [**?**], performance [**?**], fault tolerance [**?**] and security [**?**]. Our work contributes to the latter two categories. In this section we discuss related works by highlighting how they generate diversification and how they put it into practice.

There are two primary sources of software diversification: Natural Diversity and Artificial Diversity[**?**]. This work contributes to the state of the art of Artificial Diversity, which consists of synthesizing software. This thesis is founded on the work of Cohen in 1993 [**?**] as follows.

■ 2.2.1 Variants' generation

Cohen et al. proposed to generate artificial software diversification through mutation strategies. A mutation strategy is a set of rules to define how a specific component of software development should be changed to provide a different yet functionally equivalent program. Cohen et al. proposed 10 concrete transformation strategies that can be applied at fine-grained levels. All described strategies can be mixed together. They can be applied in any sequence and recursively, providing a richer diversity environment. We summarize them, complemented with the work of Baudry et al. [**?**] and the work of Jackson et al. [**?**], in 5 strategies.

[8] `https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md`

[9] `https://webassembly.org/roadmap/`

(S1) *Equivalent instructions replacement* Semantically equivalent code can replace pieces of programs. This strategy replaces the original code with equivalent arithmetic expressions or injects instructions that do not affect the computation result. There are two main approaches for generating equivalent code: rewriting rules and exhaustive searching. The replacement strategies are written by hand as rewriting rules for the first one. A rewriting rule is a tuple composed of a piece of code and a semantic equivalent replacement. For example, Cleemput et al. [**?**] and Homescu et al. [**?**] insert NOP instructions to generate statically different variants. In their works, the rewriting rule is defined as `instr => (nop instr)`, meaning that `nop` operation followed by the instruction is a valid replacement . On the other hand, exhaustive searching samples all possible programs for a specific language. In this topic, Jacob et al. [**?**] proposed the technique called superdiversification for x86 binaries. The superdiversification strategy proposed by Jacob and colleagues performs an exhaustive search of all programs that can be constructed from a specific language grammar. If one of the generated programs is equivalent to the original program, then it is reported as a variant. Similarly, Tsoupidi et al. [**?**] introduced Diversity by Construction, a constraint-based compiler to generate software diversity for MIPS32 architecture.

(S2) *Instruction reordering* This strategy reorders instructions or entire program blocks if they are independent. The location of variable declarations might change as well if compilers re-order them in the symbol tables. It prevents static examination and analysis of parameters and alters memory locations. In this field, Bhatkar et al. [**?** **?**] proposed the random permutation of the order of variables and routines for ELF binaries.

(S3) *Adding, changing, removing jumps and calls* This strategy creates program variants by adding, changing, or removing jumps and calls in the original program. Cohen [**?**] mainly illustrated the case by inserting bogus jumps in programs. Pettis and Hansen [**?**] proposed to split basic blocks and functions for the PA-RISC architecture, inserting jumps between splits. Similarly, Crane et al. [**?**] de-inline basic blocks of code as an LLVM pass. In their approach, each de-inlined code is transformed into semantically equivalent functions that are randomly selected at runtime to replace the original code calculation. On the same topic, Bhatkar et al. [**?**] extended their previous approach [**?**], replacing function calls by indirect pointer calls in C source code, allowing post binary reordering of function calls. Recently, Romano et al. [**?**] proposed an obfuscation technique for JavaScript in which part of the code is replaced by calls to complementary Wasm function.

(S4) *Program memory and stack randomization* This strategy changes the layout of programs in the host memory. Also, it can randomize how a program variant operates its memory. The work of Bhatkar et al. [**?** **?**] propose to randomize the base addresses of applications and the library memory regions in ELF binaries. Tadesse Aga and Autin [**?**] and Lee et al. [**?**] propose a technique to randomize the local stack organization for function calls using a custom LLVM compiler. Younan

et al. [**?**] propose to separate a conventional stack into multiple stacks where each stack contains a particular class of data. On the same topic, Xu et al. [**?**] transforms programs to reduce memory exposure time, improving the time needed for frequent memory address randomization.

(S5) *ISA randomization and simulation* This strategy uses a key to cypher the original program binary into another encoded binary. Once encoded, the program can be decoded only once at the target client, or it can be interpreted in the encoded form using a custom virtual machine implementation. This technique is strong against attacks involving code inspection. Kc et al. [**?**] and Barrantes et al. [**?**] proposed seminal works on instruction-set randomization to create a unique mapping between artificial CPU instructions and real ones. On the same topic, Chew and Song [**?**] target operating system randomization. They randomize the interface between the operating system and the user applications. Couroussé et al. [**?**] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. Their technique generates a different program during execution using an interpreter for their DSL. Code obfuscation [**?**] can be seen as a simplification of *ISA randomization*. The main difference between encoding and obfuscating code is that the former requires the final target to know the encoding key while the latter executes as it is in any client. Yet, both strategies are meant to tackle program analysis from potential attackers.

■ 2.2.2 Variants' equivalence

Equivalence checking between program variants is an essential component for any program transformation task, from checking compiler optimizations [**?**] to the artificial synthesis of programs discussed in this chapter. Equivalence checking proves that two pieces of code or programs are semantically equivalent [**?**]. Cohen [**?**] simplifies this checking by enunciating the following property: two programs are equivalent if given identical input, they produce the identical output. We use this same enunciation as the definition of *functional equivalence* along with this dissertation. Equivalence checking in Software Diversification aims to preserve the original functionality for programs while changing observable behaviors. For example, two programs can be statically different or have different execution times and provide the same computation.

The equivalence property is often guaranteed by construction. For example, in the case illustrated in S1 for Cleemput et al. [**?**] and Homescu et al. [**?**], their transformation strategies are designed to generate semantically equivalent program variants. However, this process is prone to developer errors, and further validation is needed. For example, the test suite of the original program can be used to check the variant. If the test suite passes for the program variant [**?**], can be considered equivalent to the original. However, it is limited due to the need for a preexisting test suite. When the test suite does not exist, another technique is needed to check for equivalence.

If there is no test suite or the technique does not inherently implement the equivalence property, the previously mentioned works use theorem solvers (SMT solvers) [**?**] to prove equivalence. For SMT solvers, the main idea is to turn the two code variants into mathematical formulas. The SMT solver checks for counter-examples. When the SMT solver finds a counter-example, there exists an input for which the two mathematical formulas return a different output. The main limitation of this technique is that all algorithms cannot be translated to a mathematical formula, for example, loops. Yet, this technique tends to be the most used for no-branching-programs checking like basic block and peephole replacements [**?**].

Another approach to check equivalence between two programs similar to using SMT solvers is by using fuzzers [**?**]. Fuzzers randomly generate inputs that provide different observable behavior. If two inputs provide a different output in the variant, the variant and the original program are not equivalent. The main limitation for fuzzers is that the process is remarkably time-expensive and requires the introduction of oracles by hand.

■ 2.2.3 Usages of Software Diversity

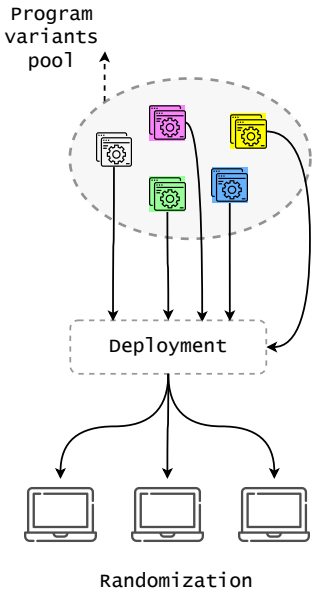
After program variants are generated, they can be used in two main scenarios: Randomization or Multivariant Execution(MVE) [**?**]. In Figure 2.2a and Figure 2.2b we illustrate both scenarios.

(U1) *Randomization:* In the context of our work *Randomization* refers to the ability of a program to be served as different variants to different clients. In the scenario of Figure 2.2a, a program is selected from the collection of variants (program's variant pool), and at each deployment, it is assigned to a random client. Jackson et al. [**?**] compare the variant's pool in Randomization with a herd immunity, since vulnerable binaries can affect only part of the client's community.

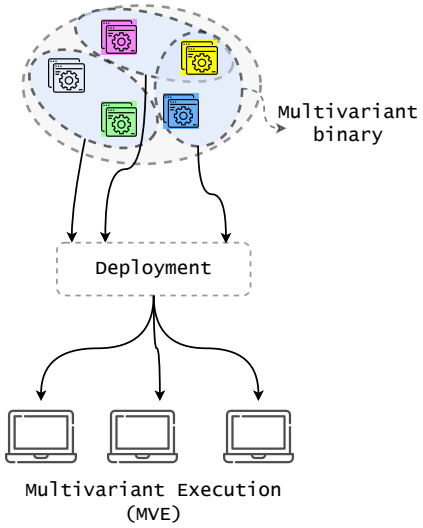
El-Khalil and colleagues [**?**] propose to use a custom compiler to generate different binaries out of the compilation process. El-Khalil and colleagues modify a version of GCC 4.1 to separate a conventional stack into several component parts, called multistacks. On the same topic, Aga and colleagues [**?**] propose to generate program variants by randomizing its data layout in memory. Their approach makes each variant to operate the same data in memory with different memory offsets. Remarkably, the Polyverse company [10] materialize randomization at the commercial level in real life. They deliver a unique Linux distribution compilation for each of its clients by scrambling the Linux packages at the source code level.

Virtual machines and operating systems can be also randomized. On this topic, Kc et al. [**?**] create a unique mapping between artificial CPU instructions and real ones. Their approach makes possible the assignation of different variants to specific target clients. Similarly, Xu et al. [**?**] recompile the Linux Kernel to

[10] `https://polyverse.com/`



(a) Randomization scenario. Given a pool of program variants, one variant is deployed per host. Each deployment randomly selects which variant is assigned to each host. The same program variant is executed in the host at every program invocation between deployments.



(b) Multivariant Execution scenario. Given a pool of program variants, a sample of the pool is packaged in a multivariant binary that is deployed. Each deployment randomly selects which multivariant binary is assigned to each host. A variant from the multivariant binary is randomly executed at runtime in the host, .

Figure 2.2: Software Diversification usages.

reduce the exposure time of persistent memory objects, increasing the frequency of address randomization.

(U2) *Multivariant Execution (MVE):* Multiple program variants are composed in one single binary (multivariant binary) [**?**]. Each multivariant binary is randomly deployed to a client. Once in the client, the multivariant binary executes its embedded program variants at runtime. Figure 2.2b illustrates this scenario.

The execution of the embedded variants can be either in parallel to check for inconsistencies or a single program to randomize execution paths [**?**]. Bruschi et al. [**?**] extended the idea of executing two variants in parallel with not-overlapping and randomized memory layouts. Simultaneously, Salamat et al. [**?**] modified a standard library that generates 32-bit Intel variants where the stack grows in

the opposite direction, checking for memory inconsistencies. Notably, Davi et al. proposed Isomeron [**?**], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. The previously mentioned works showed the benefits of exploiting the limit case of executing only two variants in a multivariant environment. Agosta et al. [**?**] and Crane et al. [**?**] used more than two generated programs in the multivariant composition, randomizing software control flow at runtime.

Both scenarios have demonstrated to harden security by tackling know vulnerabilities such as (JIT)ROP attacks [**?**] and power side-channels [**?**]. Moreover, Artificial Software Diversification is a preemptive technique for yet unknown vulnerabilities [**?**]. Our work contributes to both usages scenarios for WebAssembly.

2.3 Open challenges

In Table 2.1 we list the related work on Artificial Software Diversification discussed along with this chapter. The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order. In the following we enumerate the open challenges we have found in the literature research:

1. *WebAssembly is vulnerable:* WebAssembly is a novel technology, yet vulnerable. The adoption of defenses for it is still under development [**? ?**] and has a low pace.

2. *Software monoculture phenomenon:* WebAssembly is evolving further web browsers as a technology to support Edge-Cloud computing platforms. The Edge-Cloud computing model is based on replicating the same binary along with all computing nodes in a worldwide scale. Therefore, potential vulnerabilities are spread, highlighting a clear monoculture phenomenon [**?**].

3. *Lack of Software Diversification for WebAssembly*: Software Diversification has demonstrated to provide protection for known and yet-unknown vulnerabilities. Yet, only one software diversity approach has been applied to the context of WebAssembly [**?**]. Moreover, as we illustrate in Table 2.1, the existing works for Software Diversification do not contribute directly to WebAssembly.

4. *Lack of research on MVE for WebAssembly:* WebAssembly has a growing adoption for Edge platforms. However, researching on MVE in a distributed setting like the Edge has been less researched. Only Voulimeneas et al. [**?**] recently proposed a multivariant execution system by parallelizing the execution of the variants in different machines for the sake of efficiency. This work

5. *Porting current contributions:* The preexisting works based on the LLVM pipeline cannot be extended to Wasm due to technical challenges. The main reason is that previous works contribute to LLVM versions released before the inclusion of Wasm as an LLVM architecture. On the other hand, there is no mentioned work merging all mutation strategies in one solution.

■ Conclusions

In this chapter, we presented the background on the WebAssembly language, including its security issues and related work. This chapter aims to settle down the foundation to study automatic diversification for WebAssembly. We highlighted related work on Artificial Software Diversification, showing that it has been widely researched, not being the case for WebAssembly. On the other hand, current available implementations for Software Diversification cannot be directly ported to Wasm. The current limitations on security and the lack of software diversity approaches for WebAssembly motivate our work. We place our contributions in the field of artificial diversity. In Chapter 3 we describe the technical details that lead our contributions. Besides, the impact of our contributions is evaluated by following the methodology described in Chapter 4.

Authors	S1	S2	S3	S4	S5	U1	U2	Main technical contribution
Pettis and Hansen [?]		✓		✓		✓		Custom Pascal compiler for PA-RISC architecture
Chew and Song [?]			✓			✓		Linux Kernel recompilation.
Kc et al. [?]					✓			Linux Kernel recompilation.
Barrantes et al. [?]					✓	✓		x86 to x86 transformations using Valgrind
Bhatkar et al. [?]	✓	✓		✓		✓		ELF binary transformations
El-Khalil and Keromytis [?]						✓		custom GCC compiler for x86 architecture
Bhatkar et al. [?]	✓	✓		✓		✓		C/C++ source to source transformations and ELF binary transformations
Younan et al. [?]				✓				custom GCC compiler
Bruschi et al. [?]				✓		✓		ELF binary transformations.
Salamat et al. [?]			✓				✓	Custom GNU compiler
Jacob et al. [?]	✓	✓						x86 to x86 transformations
Salamat et al. [?]				✓			✓	x86 to x86 transformations
Amarilli et al. [?]	✓				✓	✓		Polymorphic code generator for ARM architecture
Jackson [?]	✓					✓	✓	LLVM compiler, only backend for x86 architecture
Cleemput et al. [?]	✓					✓		x86 to x86 transformations
Homescu et al. [?]	✓					✓		LLVM 3.1.0†
Crane et al. [?]	✓	✓	✓				✓	LLVM, only backend for x86 architecture
Davi et al. [?]							✓	Windows DLL instrumentation
Couroussé et al. [?]	✓	✓			✓	✓		Custom GCC compiler targeting microcontrollers
Lu et al. [?]				✓			✓	GNU assembler for Linux kernel
Belleville et al. [?]	✓			✓		✓		Only C language frontend, LLVM 3.8.0†
Aga et al. [?]				✓		✓		Data layout randomization, LLVM 3.9†
Österlund et al. [?]				✓			✓	Linux Kernel recompilation.
Xu et al. [?]				✓		✓		Custom kernel module in Linux OS
Lee et al. [?]				✓		✓		LLVM 12.0.0 backend for x86
Romano et al. [?]			✓			✓		JavaScript and Wasm intermixing

† Notice that LLVM only supports WebAssembly backend from release 7.1.0

Table 2.1: The first and second columns in the table correspond to the author names and the references to their work, followed by one column for each strategy and usage (S1, S2, S3, S4, S5, U1 and U2). The last column of the table summarizes the technical contribution and the reach of the referred work. Each cell in the table contains a checkmark if the strategy or the usage of the work match the previously mentioned classifications. The rows are sorted by the year of the work in ascending order.

03 AUTOMATIC DIVERSITY FOR WEBASSEMBLY

We aim to create artificial software diversity for WebAssembly, by providing tools to make the process easier and feasible for developers and researchers. According to our exhaustive literature review, no software provides artificial software diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies defined in Section 2.2. Our implementations are part of the contributions of this thesis. We provide two tools that complement this work: CROW and MEWE. First, the former tool generates WebAssembly program variants statically at compile time to provide randomization. The latter tool provides the tooling to generate MVE binaries for WebAssembly. In this chapter, we describe our technical contributions. In Section 3.1 we enunciate how the current state-of-the-art lead us to contribute with Software Diversification through LLVM. We follow by describing our two contributions and their main technical insights in Section 3.2 and Section 3.3. Besides, we describe a new transformation strategy as part of our contributions.

■ 3.1 Global approach

The work of Hilbig et al. [**?**] at 2021 influences our design decisions. According to their work, 70% of the WebAssembly binaries in the wild are created with LLVM-based compilers. Therefore, we provide artificial software diversity for WebAssembly through LLVM. Other solutions would have been to diversify at the source code level or the WebAssembly binary level. However, the former would limit the applicability of our work. We propose the latter for future works.

LLVM is a compound of three main components [**?**]. First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation processes improve the LLVM IR. Third and final, the backend component is in charge of generating the target machine code. In Figure 3.1 we show how we use the LLVM pipeline in our contributions, which are highlighted as dashed squares.

The global workflow in Figure 3.1 starts by receiving the source code. Then the LLVM frontend transforms it into LLVM IR representation ①. We alter the LLVM pipeline that compiles source code to Wasm by introducing a diversifier component.

The diversifier generates LLVM IR variants from the output of the frontend ②. The LLVM IR variants are inputs for our customized Wasm backend. The

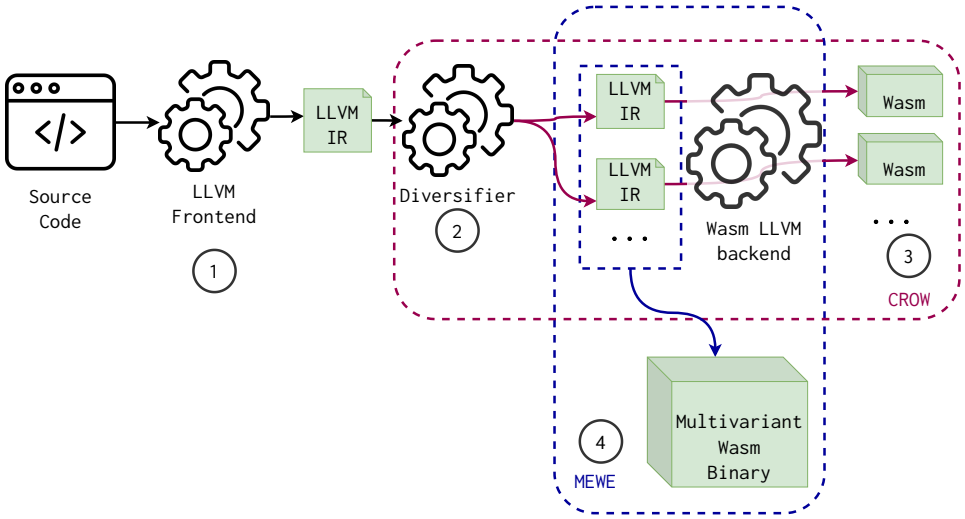


Figure 3.1: Generic workflow to create WebAssembly program variants.

diversifier and the custom Wasm LLVM backend compose CROW, which creates WebAssembly program variants out of a source code program ③. In addition, an orthogonal tool comes from the generation of LLVM IR variants at Step ②. MEWE [**?**], merges and creates multivariant binaries to provide MVE for WebAssembly ④.

■ 3.2 CROW: Code Randomization of WebAssembly bytecode

This section describes the red squared tooling in Figure 3.1 named, CROW [**?**]. CROW is a tool tailored to create semantically equivalent WebAssembly variants from an LLVM front-end output. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

In Figure 3.2, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function in its input LLVM. For each instruction, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into WebAssembly binaries. In the following, we describe our design decisions. All our implementation choices

are based on one premise: *each design decision should increase the number of WebAssembly variants that CROW creates.*

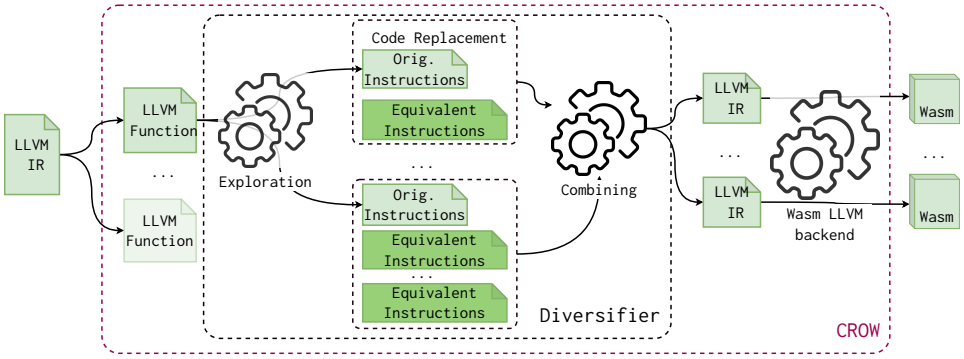


Figure 3.2: CROW components following the diagram in Figure 3.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

■ 3.2.1 Exploration

The primary component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the superdiversifier of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program that is faster or smaller than the original code while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by code size in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the superdiversifier idea of Jacob and colleagues to implement CROW because two main reasons. First, the code replacements generated by this technique outperform diversification strategies based on hand-written rules. Besides, this technique is fully automatic. Second, there is a battle-tested superoptimizer for LLVM, Souper [**?**]. This latter makes feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem

solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation the optimal replacement result of the searching. We keep more equivalent replacements during the searching by removing the halting criteria. Instead, we limit the searching for a replacement with timeout and the replacement's size. Our customized Souper reports a new code replacement as soon as an equivalent transformation is found.

Notice that the searching space exponentially increases with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions with no correspondence in the WebAssembly backend. This decision reduces the searching space. For example, creating an expression having the `freeze` LLVM instructions will increase the searching space for instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more variants.

3.2.2 Constant inferring

One code transformation strategy of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on boolean-valued variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in Section 2.2.

After a *constant inferring*, the generated program is considerably different from the original program, being suitable for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 3.1 is composed of a loop that stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 3.1: Babbage problem.

```
1   int babbage() {
2       int current = 0,
3           square;
4       while ((square=current*current) % 1000000
            ↪ != 269696) {
5           current++;
6       }
7       printf ("The number is %d\n", current);
8       return 0 ;
9   }
```

Listing 3.2: Constant inferring transformation over the original Babbage problem in Listing 3.1.

```
int babbage() {
    int current = 25264;


    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the **`while`**-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 3.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code and is removed in the final

compilation. The new program in Listing 3.2 is remarkably smaller and faster than the original code. Therefore, it offers differences both statically and at runtime. [1]

■ 3.2.3 Removing latter optimizations for LLVM

During the implementation of CROW, we have the premise of removing all built-in optimizations in the LLVM backend that could reverse Wasm variants. Therefore, we modify the WebAssembly backend in addition to the extended Souper. We disable all optimizations in the WebAssembly backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

■ 3.3 MEWE: Multi-variant Execution for WebAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE. The tool generates application-level multivariant binaries without changing the operating system or WebAssembly runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step ② in Figure 3.1 shows. CROW generates each one of these variants with fine-grained diversification at the instruction level, applying the majority of the strategies discussed in Section 2.2 and *constant inferring*. MEWE adds a new mutation strategy. It inlines function variants when appropriate, resulting in call stack diversification at runtime.

In Figure 3.3 we zoom MEWE from the blue highlighted square in Figure 3.1. MEWE takes the LLVM IR variants generated by CROW's diversifier. It then merges LLVM IR variants into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW and creates an LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, making possible the execution path randomization.

The Mixer augments the LLVM multivariant binary with a random generator. The random generator is needed to perform the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant WebAssembly binary using the same custom Wasm LLVM backend from CROW. Once generated, the generated multivariant WebAssembly binary can be deployed to any WebAssembly engine.

[1] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also, notice that the two programs in the example follow the definition of *functional equivalence* discussed in Section 2.2.

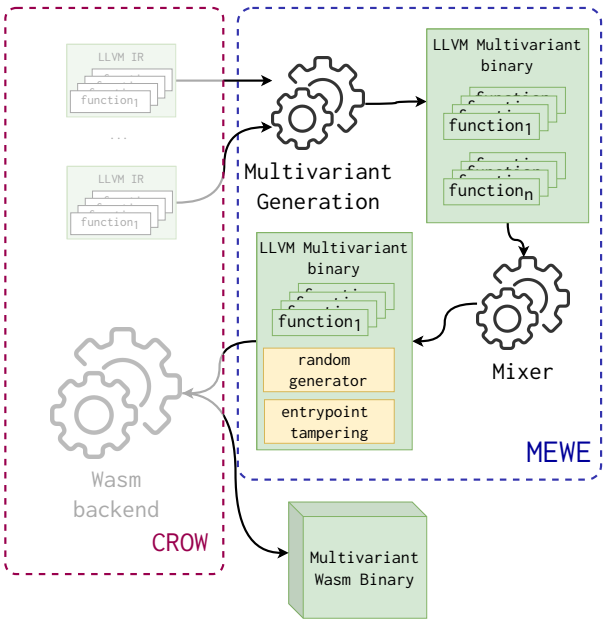


Figure 3.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates an LLVM multivariant binary composed of all the function variants. Finally, the Mixer includes the behavior in charge of selecting a variant when a function is invoked. Finally, the MEWE mixer composes the LLVM multivariant binary with a random number generation library and tampers the original application entrypoint. The final process produces a WebAssembly multivariant binary ready to be deployed.

■ 3.3.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is introducing one dispatcher function per original function with variants. A dispatcher function is a synthetic function in charge of choosing a variant at random when the original function is called. With the introduction of the dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph* $\langle N, E \rangle$ *where the nodes in* N *represent all the functions in the binary and an edge* $(f_1, f_2) \in E$ *represents a possible invocation of* f_2 *by* f_1 *[?], where the nodes are typed. The nodes in* N *have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

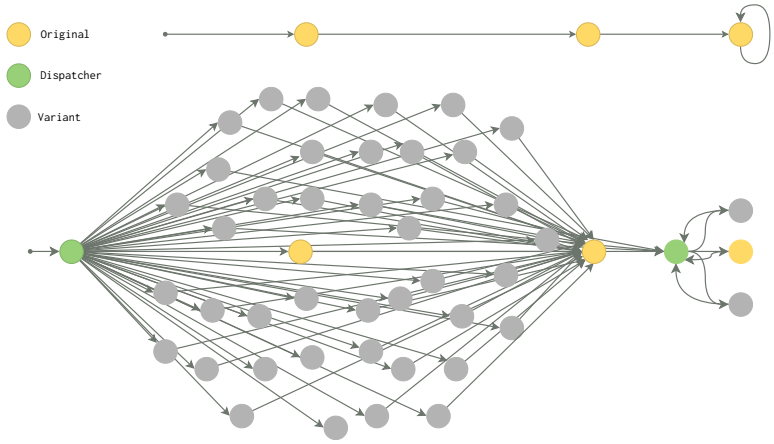


Figure 3.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

In Figure 3.4, we show the original static call graph for and original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers, and the yellow nodes are the original functions. The directed edges represent the possible calls. The original program includes three functions. MEWE generates 43 variants for the first function, none for the second, and three for the third. MEWE introduces two dispatcher nodes for the first and third functions. Each dispatcher is connected to the corresponding function variants to invoke one variant randomly at runtime.

In Listing 3.3, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3.4. It first calls the random generator, which returns a value used to invoke a specific function variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution-based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher instead of defining them again. Here we trade security over performance since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 3.3: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 3.4.

3.3.2 The Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the application's entrypoint, and merge all these components into a multivariant WebAssembly binary. We use the Rustc compiler[2] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. However, its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

3.4 Accompanying Source Code

This thesis is accompanied by the source code of both contributions, CROW and MEWE. The source code is accessible through the links:

1. CROW: `https://github.com/KTH/slumps`
2. MEWE: `https://github.com/Jacarte/MEWE`

Our software artifacts are licensed under the MIT License. The dependent source codes, such as LLVM, are licensed under their original licenses.

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

■ Conclusions

This chapter discusses the technical details of the tools implemented for our main contributions. We describe how CROW generates program variants for the sake of software diversification. We propose a global architecture for a generic LLVM superdiversifier We introduce a new mutation strategy that is a consequence of retargeting Souper as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In Chapter 4 we discuss the methodology we follow to evaluate how CROW and MEWE create software diversification.

04 METHODOLOGY

In this chapter, we present our methodology to answer the research questions enunciated in Section 1.1. We investigate three research questions. In the first question, we aim to investigate the static differences between variants. We evaluate the code properties the lead less or more software diversification. Our second research question focuses on comparing their behavior during their execution, complementing our first research question. The generated variants should be statically different, but also should provide different observable behavior. The final research question evaluates the feasibility of using the program variants in security-sensitive environments. We evaluate our generated program variants in an Edge-Cloud computing platform proposing a novel multivariant execution approach.

The main objective of this thesis is to study the feasibility of automatically creating program variants out of preexisting program sources. To achieve this objective, we use the empirical method [**?**], proposing a solution and evaluating it through quantitative analyzes in case studies. We follow an iterative and incremental approach on the selection of programs for our corpora. To build our corpora, we find a representative and diverse set of programs to generalize, even when it is unrealistic following an empirical approach, as much as possible our results. We first enunciate the corpora we share along this work to answer our research questions. Then, we establish the metrics for each research question, set the configuration for the experiments, and describe the protocol.

4.1 Corpora

Our experiments assess the impact of artificially created diversity. The first step is to build a suitable corpus of programs' seeds to generate the variants. Then, we answer all our research questions with three corpora which follow two main properties: 1) *functionally diverse:* the selection of the programs is not biased by functionally fixed tasks, for example, the programs in one of our corpora solve from the *Babbage* problem to *Convex Hull* calculation; and 2) *representative:* our corpora have 3021 programs that can be ported to WebAssembly, representing approximately 40% of the unique Wasm binaries in the wild [**?**].

We build our three corpora in an escalating strategy based on the merging of our previous publications. The first corpus is diverse and contains simple programs in terms of code size, making them easy to manually analyze. The second corpus is a project meant for security-sensitive applications. The third corpus is a QR encoding

decoding algorithm. In the following, we describe the filtering and description of each corpus.

1. **Rosetta**: We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains many tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from the Rosetta Code, representing 989 programs as of 01/26/2020. We then apply several filters: the programs should successfully compile and, they should not require user inputs to automatically execute them, the programs should terminate and should not result in non-deterministic results.

   The result of the filtering is a corpus of 303 C programs. All programs include a single function in terms of source code. These programs range from 7 to 150 lines of code.

2. **Libsodium**: This project is encryption, decryption, signature, and password hashing library implemented in 102 separated modules. The modules have between 8 and 2703 lines of code per function. This project is selected based on two main criteria: first, its importance for security-related applications, and second, its suitability to collect the modules in LLVM intermediate representation.

3. **QrCode**: This project is a QrCode and MicroQrCode generator written in Rust. This project contains 2 modules having between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for collecting the modules in their LLVM representation. Remarkably, this project increases the complexity of the previously selected projects due to its integration with the generation of images.

In Table 4.1 we listed the corpus name, the language of the programs in the corpus, the number of modules, the total number of functions, the range of lines of code, and the original location of the corpus.

■ 4.2 RQ_1. To what extent can we artifically generate program variants for WebAssembly?

This research question investigates whether we can artificially generate program variants for WebAssembly. We use CROW to generate variants from an original program, written in C/C++ in the case of Rosetta corpus and LLVM bitcode modules in the case of Libsodium and QrCode. In Figure 4.1 we illustrate the workflow to generate WebAssembly program variants. We pass each function of

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

Corpus	Lang.	No. modules	No. functions	LOC range	Location
Rosetta	C	-	303	7 - 150	`https://github.com/KTH/slumps/tree/master/benchmark_programs/rossetta/valid/no_input`
Libsodium	LLVM IR + Rust	102	869	8 - 2703	`https://github.com/jedisct1/libsodium/tree/2b5f8f2b6810121c2d9a8cc8a392e01f4d3de433`
QrCode	LLVM IR + Rust	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust/commit/faa4397ba7c5f441cb9a2b436c1e84a0d52ae942`
Total			3021		

Table 4.1: Corpora description. The table is composed by the name of the corpus, programming language of the programs in the corpus, the number of modules, the number of functions, the lines of code range and the location of the corpus.

the corpora to CROW as a program to diversify. To answer RQ1, we study the outcome of this pipeline, the generated WebAssembly variants.

Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of variants and the number of unique variants for each original function of each corpus. On top, we define the aggregation of these former two values to quantitatively evaluate RQ1 at the corpus level.

We start by defining what a program's population is. This definition can be applied in general to any collection of variants of the same program. All definitions are based upon bytecodes and not the source code of the programs.

Definition 2. *Program's population $M(P)$:* Given a program P and its generated variants v_i, the program's population is defined as.

$$M(P) = \{v_i \text{ where } v_i \text{ is a variant of P}\}$$

Notice that, the program's population includes the original program P.

Beyond the program's population, we also want to compare how many program variants are unique. The subset of unique programs in the program's population hints how the variants are different between them and not only against the original

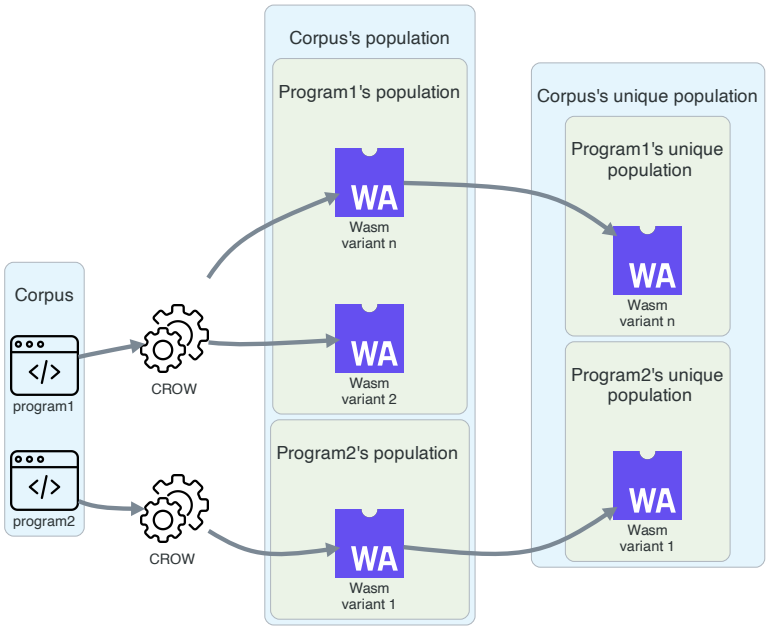


Figure 4.1: The program variants generation for RQ1.

program. For example, imagine a program P with two program variants V_1 and V_2, the program population is composed by $\{P, V_1$ and $V_2\}$, where V_1 is different from P, and V_2 is different from P. Either, if V_1 is equal or different from V_2, the program's population still be the same.

Definition 3. *Program's unique population $U(P)$:* Given a program P and its program's population $M(P)$, the program's unique population is defined as.

$$U(P) = \{v \ \in \ M(P)\}$$

such that $\forall v_i, v_j \in U(P)$, $md5sum(v_i) \neq md5sum(v_j)$. $Md5sum(v)$ is the md5 hash calculated over the bytecode stream of the program file v. Notice that, the original program P is included in $U(P)$.

Metric 1. *Program's population size $S(P)$:* Given a program P and its program's population $M(P)$ according to Definition 2, the program's population size is defined as.

$$S(P) = |M(P)|$$

Metric 2. *Program's unique population size $US(P)$:* Given a program P and its program's unique population $U(P)$ according to Definition 3, the program's unique population size is defined as.

$$US(P) = |U(P)|$$

Metric 3. *Corpus population size $CS(C)$:* Given a program's corpus C, the corpus population size is defined as the sum of all program's population sizes over the corpus C.

$$CS(C) = \Sigma S(P) \ \forall \ P \ \in \ C$$

Metric 4. *Corpus unique population size $UCS(C)$:* Given a program's corpus C, the corpus unique population size is defined as the sum of all program's unique population sizes over the corpus C

$$UCS(C) = \Sigma US(P) \ \forall \ P \ \in \ C$$

■ Protocol

To generate program variants, we synthesize program variants with an enumerative strategy, checking each synthesis for equivalence modulo input [**?**] against the original program, as it is described in Section 3.2. For obvious reasons, this space is nearly impossible to explore in a reasonable time as soon as the limit of instructions increases. Therefore, we use two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. In our experiments for RQ1, we use all instructions in the CROW diversifier synthesis.

The former parameter allows us to find a trade-off between the number of variants that are synthesized and the time taken to produce them. For the current evaluation, given the size of the corpus and the properties of its programs, we set the exploration time to 1 hour maximum per function for Rosetta. In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function. The decision behind the usage of lower timeout for Libsodium and QrCode is motivated by the properties listed in Table 4.1. The latter two corpora are remarkably larger regarding the number of instructions and functions count.

We pass each of the $303 + 869 + 1849$ functions in the corpora to CROW, as Figure 4.1 illustrates, to synthesize program variants. We calculate the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus and conclude by answering RQ1.

■ 4.3 RQ_2. To what extent are the generated variants dynamically different?

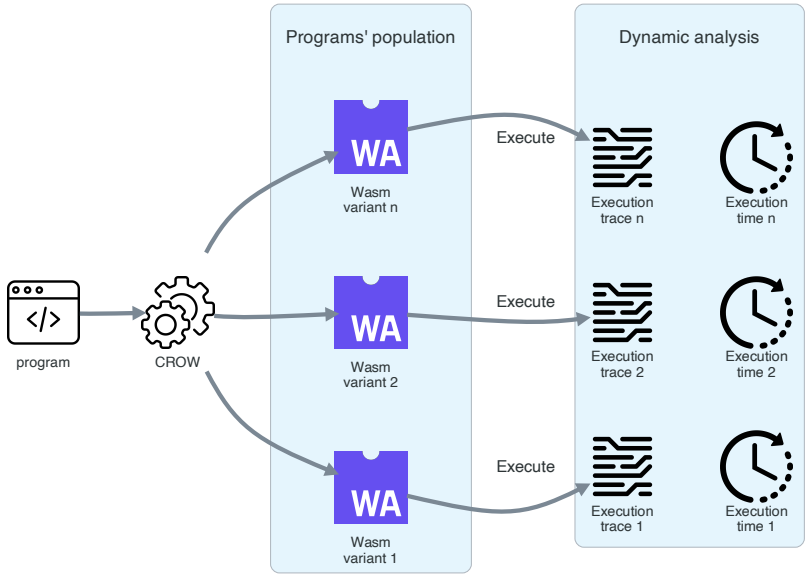


Figure 4.2: Dynamic analysis for RQ2.

In this second research question, we investigate to what extent the artificially created variants are dynamically different between them and the original program. To conduct this research question, we could separate our experiments into two fields as Figure 4.2 illustrates: static analysis and dynamic analysis. The static analysis focuses on the appreciated differences among the program variants, as well as between the variants and the original program. We perform the static analysis in answering RQ1 in Section 4.2. With RQ2, we focus on the last category, the dynamic analysis of the generated variants. This decision is supported because dynamic analysis complements RQ1 and, it is essential to provide a full understanding of diversification. We use the original functions from Rosetta corpus described in Section 4.1 and their variants generated to answer RQ1. We use only Rosetta to answer RQ2 because this corpus is composed of simple programs that can be executed directly without user interaction, *i.e.,* we only need to call the interpreter passing the WebAssembly binary to it. To dynamically compare

programs and their variants, we execute each program on each programs' population to collect and execution times. We define execution trace and execution time in the following section.

■ Metrics

We compare the execution traces of two any programs of the same population with a global alignment metric. We propose a global alignment approach using Dynamic Time Warping (DTW). Dynamic Time Warping [**?**] computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is a distance metric. The larger the DTW distance, the more different the two sequences are. DTW has been used for comparing traces in different domains. For software, De A. Maia et al. [**?**] proposed to identify similarity between programs from execution traces. As we discussed in Section 2.1, a theoretical WebAssembly engine perform `push` and `pop` operations when the program instructions are executed. Therefore, in our experiments, we define the execution traces as the sequence of the stack operations during the execution of the WebAssembly program. In the following, we define the $TraceDiff$ metric.

Metric 5. *TraceDiff:* Given two programs P and P' from the same program's population, TraceDiff(P,P'), computes the DTW distance collected during their execution.
A TraceDiff of 0 means that both traces are identical. The higher the value, the more different the traces.

Moreover, we use the execution time distribution of the programs in the population to complement the answer to RQ2. For each program pair in the programs' population, we compare their execution time distributions. We define the execution time as follows:

Metric 6. *Execution time:* Given a WebAssembly program P, the execution time is the time spent to execute the binary.

■ Protocol

To compare program and variants behavior during runtime, we analyze all the unique program variants generated to answer RQ1 in a pairwise comparison using the value of aligning their execution traces (Metric 5). We use SWAM[2] to execute each program and variant to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions, including the virtual stack operations.

[2] `https://github.com/satabin/swam`

Furthermore, we collect the execution time, Metric 6, for all programs and their variants. We compare the collected execution time distributions between programs using a Mann-Withney U test [**?**] in a pairwise strategy.

■ 4.4 RQ_3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

TODO The last method is too short

To answer RQ3, we use the variants generated for the programs of Libsodium and QrCode corpora, we take $2 + 5$ programs interconnecting the LLVM bitcode modules (mentioned in Table 4.1). We illustrate the protocol to answer RQ3 in Figure 4.3 starting from the creation of the programs' population.

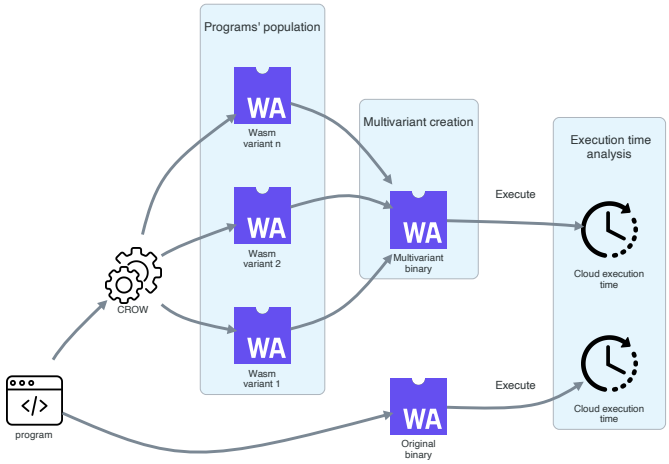


Figure 4.3: Multivariant binary creation and workflow for RQ3 answering.

In RQ3, we study whether the created variants can be used in real-world applications and what properties offer the composition of the variants as multivariant binaries. We build multivariant binaries (according to Definition 1), and we deploy and execute them at the Edge. The usage of Edge-Cloud computing platforms to answer RQ3 is motivated by two reasons. First, it is an emerging technology. Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [**? ?**]. This has encouraged edge computing platforms like Cloudflare and Fastly to use WebAssembly to deploy client applications in a modular and sandboxed manner [**? ?**]. Second, Edge-Cloud computing platforms are shown to be not completely secure [**?**] and

multivariant execution offers a preemptive technique against predictable behaviors such as execution time.

Metrics

To answer RQ3, we build multivariant WebAssembly binaries (see Definition 1) meant to provide execution path randomization. We use the execution time of the multivariant binaries to answer RQ3. We use the same metric defined in Metric 6 for the execution time of multivariant binaries.

Protocol

We answer RQ3 by analyzing a real-world scenarios on the Edge. Edge applications are designed to be deployed as isolated HTTP services, having one single responsibility that is executed at every HTTP request. This development model is known as serverless computing, or function-as-a-service [**? ?**]. We deploy and execute the multivariant binaries as end-to-end HTTP services on the Edge, and we collect their execution times. To remove the natural jitter in the network, the execution times are measured at the backend space, *i.e.,* we collect the execution times inside the Edge node and not from the client computer. Therefore, we instrument the binaries to return the execution time as an HTTP header.

We do the collection of the execution times twice, for the original program and its multivariant binary. We deploy and execute the original and the multivariant binaries on 64 edge nodes located around the world. In Figure 4.4 we illustrate the word wide location of the edges nodes.

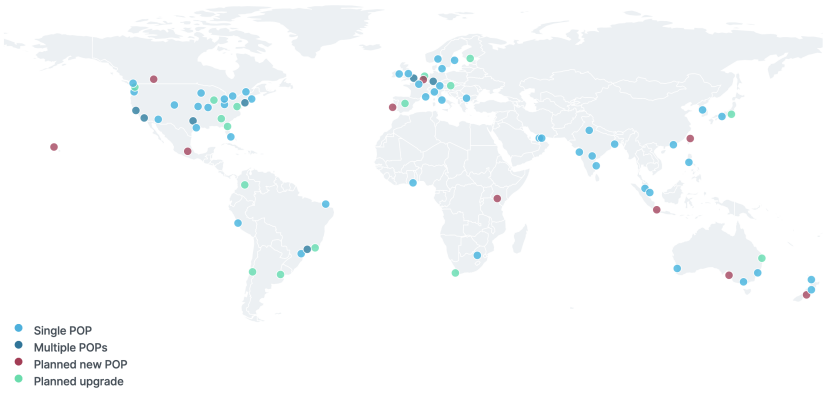


Figure 4.4: Screenshot taken from the Fastly Inc. platform used in our experiments for RQ3. Blue and darker blue dots represent the edge nodes used in our experiments.

We collect 100k execution times for each binary, both the original and multivariant binaries. The number of execution time samples is motivated by the seminal work of Morgan et al. [**?**]. We perform a Mann-Withney U test [**?**] to compare both execution time distributions. If the P-value is lower than 0.05, the two compared distributions are different.

■ Conclusions

This chapter presents the methodology we follow to answer our three research questions. We first describe and propose the corpora of programs used in this work. We propose to measure the ability of our approach to generate variants out of 3021 functions of our corpora. Then, we suggest using the generated variants to study to what extent they offer different observable behavior through dynamic analysis. We propose a protocol to study the impact of the composition variants in a multivariant binary deployed at the Edge. Besides, we enumerate and enunciate the properties and metrics that might lead us to answer the impact of automatic diversification for WebAssembly programs. In the next chapter, we present and discuss the results obtained with this methodology.

05 RESULTS

In this chapter, we sum up the results of the research of this thesis. We illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 4.

5.1 RQ_1. To what extent can we artifically generate program variants for WebAssembly?

As we describe in Section 4.2, our first research question aims to answer how to artifically generate WebAssembly program variants. This section is organized as follows. First we present the general results calculating the *Corpus population size*(Metric 3) and *Corpus unique population size*(Metric 4) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

5.1.1 Program's populations

We summarize the results in Table 5.1. The table illustrates the corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant), the cumulative number of variants (*Corpus population size*) and the cumulative number of unique variants (*Corpus unique population size*).

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a diversification timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, representing a 11.78% of the total. As the four and fifth columns show, the number of artifically created variants and the number of unique variants are larger than the original number of functions by one order of magnitude. In the case of Rosetta, the corpus population size is close to one million of programs. The remarkable difference between the total number of variants and the number

of unique variants (fourth and fifth columns) is mainly due to the *replacements combining* process discussed in Section 3.2.
TODO M: add histogram on variant sizes

Corpus	#Functions	# Diversified	# Variants	# Unique Variants
Rosetta		239	809900	2678
Libsodium	869	85	4272	3805
QrCode	1849	32	6369	3314
	3021	356	820541	9797

Table 5.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the cumulative number of generated variants and the cumulative number of unique variants.

5.1.2 Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 5.1). Our approach successfully diversified 239/303, 85/869 and 32/1849 of the original functions for Rosetta, Libsodium and QrCode respectively. The main reason of this phenomenon is the set timeout for CROW.

We have noticed a remarkable difference between the number of diversified functions for each corpus, 809900 programs for Rosetta 4272 for Libsodium and 6369 for QrCode. The corpus population size for Rosetta is two orders of magnitude larger compared to the other two corpora. The reason behind the large number of variants for Rosetta is that, after certain time, our approach starts to combine the code replacements to generate new variants. However, looking at the fifth column, the number of unique variants have the same order of magnitude for all corpora. The variants generated out of the combination of several code replacements are not necessarily unique. Some code replacements can dominate over others, generating the same WebAssembly programs.

A low timeout offers more unique variants compared to the population size despite the low number of diversified functions, like the Libsodium and QrCode cases. This happens because, CROW first generates variants out of single code replacements and then starts to combine them. Thus, more unique variants are generated in the very first moments of the diversification process with CROW.

Apart from the timeout and the combination of variants phenomenon, we manually analyze programs, searching for properties attempting to the generation of program variants using CROW. As we previously mentioned in Section 3.2, *constant*

inferring is a new contribution of ours to the collection of Software Diversification strategies enumerated in Section 2.2. We have observed that our approach searches for a constant inferring for more than 45% of the instructions of each function while constant values cannot be inferred in all cases. The main reason is that memory operations are also included into the inferring while our tool is oblivious to a memory model, making unsuccessful the constant replacement.

5.1.3 Properties for large diversification

We manually analyzed the programs to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 5.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 5.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 5.3), using the intrinsics of the underlying computation model.

Listing 5.1: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 5.2: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

Listing 5.3: Diversification through overflow of integer operands.

```
i32.const 2
i32.mul
```

```
i32.const 2
i32.mul
i32.const
  -2147483647
i32.mul
```

At the WebAssembly level, we have not observed variants performing changes in the control flow structure of the variants (S3). We manually analyze the machine code generated by V8 (as it was discussed in Section 2.1). We have observed that, for different variants, we are changing the number of jumps and its location inside the machine code.

Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. Constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants.

■ 5.2 RQ_2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated to answer RQ1 for Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 5) and the execution time distributions (Metric 6) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with TraceDiff of Metric 5. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions Metric 6, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

■ 5.2.1 Stack operation traces.

In Figure 5.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 5) for a program of Rosetta corpus for which we generate variants. Each dot represents a comparison between two programs' traces according to Metric 5. The programs are sorted by their number of variants in descending order. For the sake of illustration, we filter out those programs for which we generate only 2 unique variants.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of TraceDiff is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which TraceDiff value is zero, *i.e.,* they result in the same stack operation trace. We

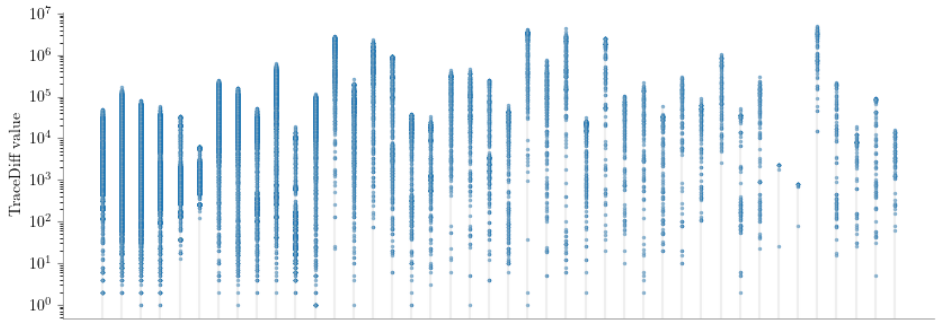


Figure 5.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 5.

identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

```
(1) i32.lt_u      i32.lt_s
(2) i32.le_s      i32.lt_u

(3) i32.ne        i32.lt_u
(4) local.get 6   local.get 4
```

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

■ 5.2.2 Execution times.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution

times for the generated variants. Besides the execution times of programs can be used by malicious clients to construct personalized attacks [**?**]. Therefore, by measuring the execution times, we assess the diversification of observable behaviors evaluated in real-world security scenarios.

For each program's population, we compare the execution time distributions, Metric 6, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs' population of Rosetta. The plots in Figure 5.2 show the execution time distributions for the `Hilbert curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 5.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 5.3.

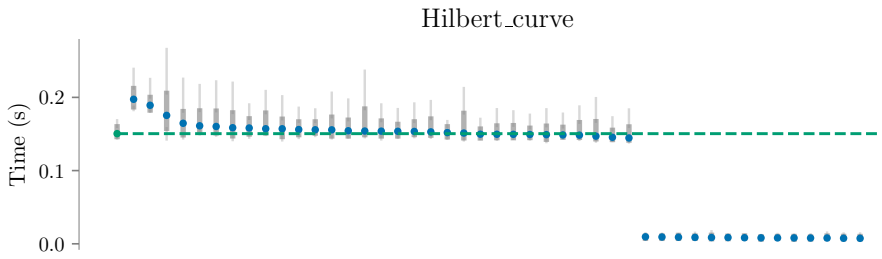


Figure 5.2: Execution time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. Concretely, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a resilient execution.

■ 5.3 RQ_3. To what extent do the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for Libsodium and QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

5.3.1 Execution times

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 5.3, each subplot represents the quantile-quantile plot [**?**] of the two distributions, original and multivariant binary. This kind of plots is used to compare the shapes of distributions, providing a graphical comparison of location, scale, and skewness for two distributions. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. The distribution for the multivariant binary is different and even more spread than the original one. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct.

Conclusions

Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a

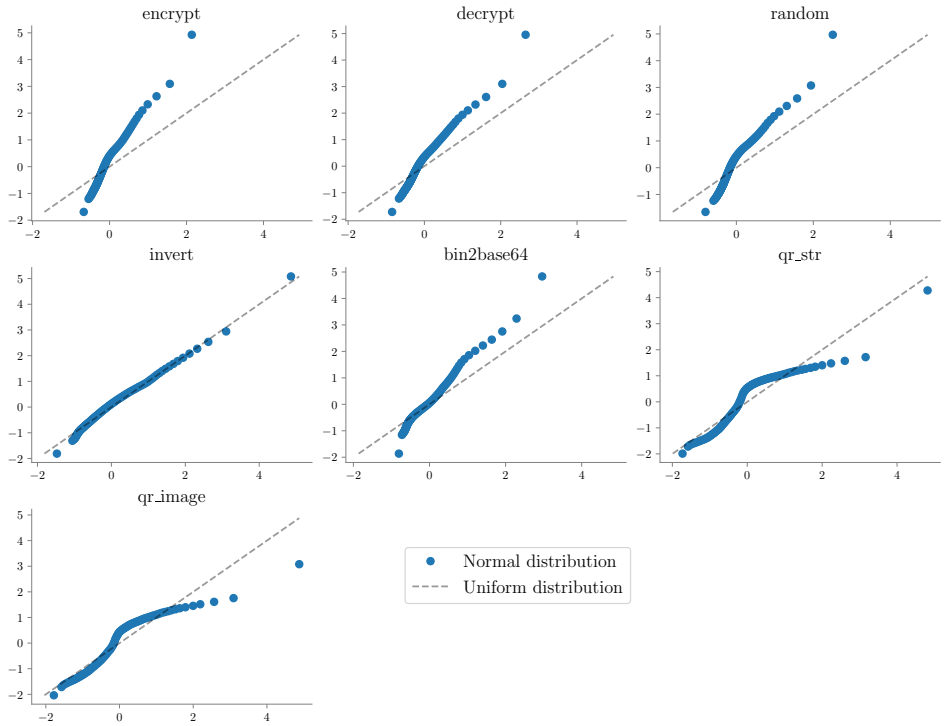


Figure 5.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

06 CONCLUSION AND FUTURE WORK

WebAssembly has become a new technology for web browsers and standalone engines such as the ones used in Edge-Cloud platforms. WebAssembly is designed with security and sandboxing premises, yet, is still vulnerable. Besides, since it is a relatively new technology, new vulnerabilities appear in the wild faster than the adoption of patches and defenses. As a widely studied field, software diversification could be a solution for known and yet-unknown vulnerabilities. Yet, there is no research on this field for WebAssembly.

We propose an automatic approach to generate software diversification for WebAssembly in this work. In addition, we provide complementary implementation for our approaches, including a generic LLVM superdiversifier that potentially extends our ideas to other programming languages. We empirically demonstrate the impact of our approach by providing Randomization and Multivariant Execution (MVE) for WebAssembly. For this, we provide two tools, CROW and MEWE. CROW completely automatizes the process by using a superdiversifier. Besides, MEWE provides execution path randomization for an MVE. This chapter is organized into two sections. In Section 6.1, we summarize the main results we found by answering our research questions enunciated in Chapter 1. Finally, Section 6.2 describes potential future work that could extend this dissertation.

6.1 Summary of the results

We enunciate the three research questions in Chapter 1. With the first research question, we investigate the static properties of the software diversification for WebAssembly generated by our approaches. We answer our first research question by creating programs variants for 3021 original programs. With CROW, we create program variants for the 11.78% of the programs in our corpora. We study the properties of the generated variants at the level of generated programs' population. Thus, we identify the challenges that attempt against the generation of unique program variants. Besides, we highlight the code properties that offer numerous program variants.

Complementary with our first research question, we evaluate the dynamic properties of the program variants generated to answer our first research question. We execute each of the 303 original programs and its generated variants for the Rosetta. For each execution, we collect their execution trace and their execution times. We demonstrate that the WebAssembly variants generated by CROW offer

remarkably different execution traces. Similarly, the execution times are different between each program and its variants. For the 71% of the diversified programs, at least one variant has an execution time distribution that is different from the original program's execution time distribution. Moreover, CROW generates both faster and slower variants. Nevertheless, we highlighted the importance of dynamic analysis for software diversification.

Our last and third research question evaluates the impact of providing a worldwide MVE for WebAssembly. We use MEWE to build multivariant binaries for the program variants generated for Libsodium and QrCode corpora. We deploy the generated multivariant binaries in an Edge-Cloud platform, collecting their execution times. The addition of runtime path randomization to multivariant binaries provides significant differences between the execution of the original binary and the multivariant binary. The observed differences lead us to conclude that no specific variant can be inferred from studying the execution time of the multivariant binaries. Therefore, attacks that rely on measuring precise execution times are more challenging to conduct.

Overall, these results show that our approaches can provide an automated end-to-end solution for the diversification of WebAssembly programs. Our approaches harden observable properties commonly used to conduct attacks, such as static code analysis, execution traces, and execution time. Therefore, our approaches harden unknown and yet-unknown vulnerabilities.

6.2 Future work

Along with this dissertation, we highlighted challenges and limitations. In all cases, we proposed solutions, yet, some of them could be explored more in-depth as a call for optimization. For example, as we mentioned in Section 5.1 our solution provides program variants but remarkably lower unique variants as a consequence of the replacement combining process of CROW (Section 3.2). Techniques relying on intelligent heuristics could help increase the generation of unique variants by early discarding unsound combinations. On the other hand, constant inferring does not always finish in a successful replacement due to the CROW's obliviousness to some computation models, such as memory operations. A solution could also be to use heuristics to select which part of the code is more probable to become a constant assignment.

As we mentioned in Chapter 3, another approach to providing software diversification for Wasm could be binary to binary transformations. This approach could be used to increase resilience in malware classifiers through the study of diversification as an obfuscation technique [**?**]. Obfuscation of Wasm code could be used to measure the accuracy of malware classifiers.

By using a superdiversifier, the generated code transformations outperform hand-written transformations. Evidence of this is the CVE[1] found in the code

[1] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-befo`

generation component of wasmtime. We found this CVE during the implementation of MEWE with one of the generated variants. This highlighted the need for better strategies for stressing compilers, interpreters, and validators of Wasm. CROW and MEWE might improve fuzzing campaigns [**?**], preventing vulnerabilities by providing better testing.

`re-it-became-a-problem`

Part II

Included papers

SUPEROPTIMIZATION OF WEBASSEMBLY BYTECODE

Javier Cabrera-Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, Martin Monperrus
Programming 2020, MoreVMs'20

Superoptimization of WebAssembly Bytecode

Javier Cabrera Arteaga
KTH
Sweden
javierca@kth.se

Shrinish Donde
KTH
Sweden
shrinish@kth.se

Jian Gu
KTH
Sweden
jiagu@kth.se

Orestis Floros
KTH
Sweden
forestis@kth.se

Lucas Satabin
Mobimeo
Germany
lucas.satabin@gnieh.org

Benoit Baudry
KTH
Sweden
baudry@kth.se

Martin Monperrus
KTH
Sweden
martin.monperrus@csc.kth.se

ABSTRACT

Motivated by the fast adoption of WebAssembly, we propose the first functional pipeline to support the superoptimization of WebAssembly bytecode. Our pipeline works over LLVM and Souper. We evaluate our superoptimization pipeline with 12 programs from the Rosetta code project. Our pipeline improves the code section size of 8 out of 12 programs. We discuss the challenges faced in superoptimization of WebAssembly with two case studies.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; *Retargetable compilers*; Software implementation planning.

KEYWORDS

superoptimization, webassembly, web, optimization, lllvm

ACM Reference Format:
Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. 2020. Superoptimization of WebAssembly Bytecode. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3397567

1 INTRODUCTION

After HTML, CSS, and JavaScript, WebAssembly (WASM) has become the fourth standard language for web development [7]. This new language has been designed to be fast, platform-independent, and experiments have shown that WebAssembly can have an overhead as low as 10% compared to native code [11]. Notably, WebAssembly is developed as a collaboration between vendors and has been supported in all major browsers since 2017.

The state-of-art compilation frameworks for WASM are Emscripten and LLVM [5, 6], they generate WASM bytecode from high-level languages (e.g. C, C++, Rust). These frameworks can apply a sequence of optimization passes to deliver smaller and faster binaries. In the web context, having smaller binaries is important, because they are delivered to the clients over the network, hence smaller binaries means reduced latency and page load time. Having smaller WASM binaries to reduce the web experience is the core motivation of this paper.

To reach this goal, we propose to use superoptimization. Superoptimization consists of synthesizing code replacements in order to further improve binaries, typically in a way better than the best optimized output from standard compilers [4, 15]. Given a program, superoptimization searches for alternate and semantically equivalent programs with fewer instructions [12]. In this paper, we consider the superoptimization problem stated as finding an equivalent WebAssembly binary such that the size of the binary code is reduced compared to the default one.

This paper presents a study on the feasibility of superoptimization of WebAssembly bytecode. We have designed a pipeline for WASM superoptimization, done by tailoring and integrating open-source tools. Our work is evaluated by building a benchmark of 12 programs and applying superoptimization on them. The pipeline achieves a median size reduction of 0.33% in the total number of WASM instructions.

To summarize, our contributions are:

- The design and implementation of a functional pipeline for the superoptimization of WASM.
- Original experimental results on superoptimizing 12 C programs from the Rosetta Code corpus.

2 BACKGROUND

2.1 WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. As described in the WebAssembly Core Specification [7], WebAssembly is a portable, low-level code format designed for efficient execution and compact representation. WebAssembly has been first announced publicly in 2015. Since 2017, it has been implemented by four major web browsers (Chrome, Edge, Firefox, and Safari). A paper by Haas et al. [11] formalizes the language and its type system, and explains the design rationale.

The main goal of WebAssembly is to enable high performance applications on the web. WebAssembly can run as a standalone VM or in other environments such as Arduino [10]. It is independent of any specific hardware or languages and can be compiled for

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3397567

modern architectures or devices, from a wide variety of high-level languages. In addition, WebAssembly introduces a memory-safe, sand-boxed execution environment to prevent common security issues, such as data corruption and security breaches.

Since version 8, the LLVM compiler framework supports the WebAssembly compilation target by default [6]. This means that all languages that have an LLVM front end can be directly compiled to WebAssembly. Binaryen [14], a compiler and toolchain infrastructure library for WebAssembly, supports compilation to WebAssembly as well. Once compiled, WASM programs can run within a web browser or in a standalone runtime [10].

2.2 Superoptimization

Given an input program, code superoptimization focuses on *searching* for a new program variant which is faster or smaller than the original code, while preserving its correctness [2]. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [12] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. However, for larger instruction sets, the exhaustive exploration approach becomes virtually impossible. Because of this, the paper proposes a pruning method over the search space and a fast probabilistic test to check programs equivalence.

State of the art superoptimizers such as STOKE [16] and Souper [15] make modifications to the code and generate code rewrites. A cost function evaluates the correctness and performance of the rewrites. Correctness is generally estimated by running the code against test cases (either provided by the user or generated automatically, e.g. symbolic evaluation on both original and replacement code).

2.3 Souper

Souper is a superoptimizer for LLVM [15]. It enumerates a set of several optimization candidates to be replaced. An example of such a replacement is the following, replacing two instructions by a constant value:

```
%0:i32 = var (range=[1,0))
%1:i1 = ne 0:i32, %0
```

```
cand %1 1:i1
```

In this case, Souper finds the replacement for the variable `%1` as a constant value (in the bottom part of the listing) instead of the two instructions above.

Souper is based on a Satisfiability Modulo Theories (SMT) solver. SMT solvers are useful for both verification and synthesis of programs [8]. With the emergence of fast and reliable solvers, program alternatives can be efficiently checked, replacing the probabilistic test of Massalin [12] as mentioned in subsection 2.2.

In the code to be optimized, Souper refers to the optimization candidates as *left-hand side* (LHS). Each LHS is a fragment of code that returns an integer and is a target for optimization. Two different LHS candidates may overlap. For each candidate, Souper tries to find a *right-hand side* (RHS), which is a fragment of code that is combined with the LHS to generate a replacement. In the original paper's benchmarks [15], Souper optimization passes were found to further improve the top level compiler optimizations (-O3 for *clang*, for example) for some programs.

Souper is a platform-independent superoptimizer. The cost function is evaluated on an intermediate representation and not on the code generated for the final platform. Thus, the tool may miss optimizations that make sense for the target instruction set.

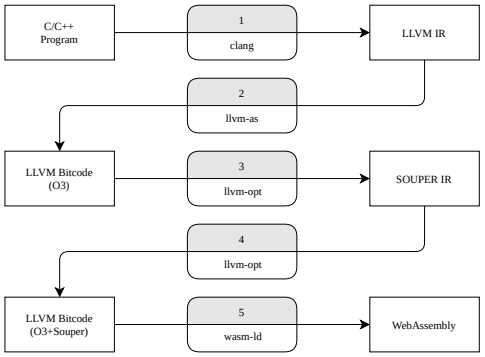


Figure 1: Superoptimization pipeline for WebAssembly based on Souper.

3 WASM SUPEROPTIMIZATION PIPELINE

The key contribution of our work is a superoptimization pipeline for WebAssembly. We faced two challenges while developing this pipeline: the need for a correct WASM generator, and the usage of a full-fledged superoptimizer. The combination of the LLVM WebAssembly backend and Souper provides the solution to tackle both challenges.

3.1 Steps

Our pipeline is a tool designed to output a superoptimized WebAssembly binary file for a given C/C++ program that can be compiled to WASM. With our pipeline, users write a high level source program and get a superoptimized WebAssembly version.

The pipeline (illustrated in Figure 1) first converts a high-level source language (e.g. C/C++) to the LLVM intermediate representation (LLVM IR) using the Clang compiler (Step 1). We use the code generation options in clang in particular the -O3 level of optimization which enables aggressive optimizations. In this step, we make use of the LLVM compilation target for WebAssembly 'wasm32-unknown-unknown'. This flag can be read as follows: wasm32 means that we target the 32 bits address space in WebAssembly; the second and third options set the compilation to any machine and performs inline optimizations with no specific strategy. LLVM IR is emitted as output.

Secondly, we use the LLVM assembler tool (llvm-as) to convert the generated LLVM IR to the LLVM bitcode file (Step 2). This LLVM assembler reads the file containing LLVM IR language, translates it to LLVM bitcode, and writes the result into a file. Thus, we make use of the optimizations from clang and the LLVM support for WebAssembly before applying superoptimization to the generated code.

Next, we use Souper, discussed in subsection 2.3, to add further superoptimization passes. Step 3 generates a set of optimized candidates, where a candidate is a code fragment that can be optimized by Souper. From this, Souper carries out a search to get shorter instruction sequences and uses an SMT solver to test the semantic equivalence between the original code snippet and the optimized one [15].

Step 4 produces a superoptimized LLVM bitcode file. The **opt** command is the LLVM analyzer that is shipped with recent LLVM versions. The purpose of the **opt** tool is to provide the capability of adding third party optimizations (plugins) to LLVM. It takes LLVM source files and the optimization library as inputs, runs the specified optimizations and outputs the optimized file or the analysis results. Souper is integrated as a specific pass for LLVM **opt**.

The last step of our pipeline consists of compiling the generated superoptimized LLVM bitcode file to a WASM program (Step 5). This final conversion is supported by the WebAssembly linker (wasm-ld) from the LLD project [13]. wasm-ld receives the object format (bitcode) that LLVM produces when run with the 'wasm32-unknown-unknown' target and produces WASM bytecode.

To our knowledge, this is the first successful integration of those tools into a working pipeline for superoptimizing WebAssembly code.

3.2 Insights

We note that Souper has been primarily designed with the LLVM IR in mind and requires a well-formed SSA representation of the program under superoptimization. The biggest challenge with WebAssembly is that there no complete transformation from WASM to SSA. In our pipeline, we work around this by assuming we have access to source code, this alternative path may be valid for plugging other binary format into Souper.

4 EXPERIMENTS

To study the effects and feasibility of applying superoptimization to WASM code, we run the superoptimization pipeline on a benchmark of programs.

The benchmark is based on the Rosetta Code corpus[1]. We have selected 12 C language programs that compile to WASM. Our selection of the programs is based on the following criteria:

(1) The programs can be successfully compiled to LLVM IR.
(2) They are diverse in terms of application domain.
(3) The programs are small to medium sized: between 15 and 200 lines of C code each.
(4) They have no dependencies to external libraries.

The code of each program is available as part of our experimental package[2].

[1] http://rosettacode.org
[2] https://github.com/KTH/slumps/tree/master/utils/pipeline/benchmark4pipeline_c

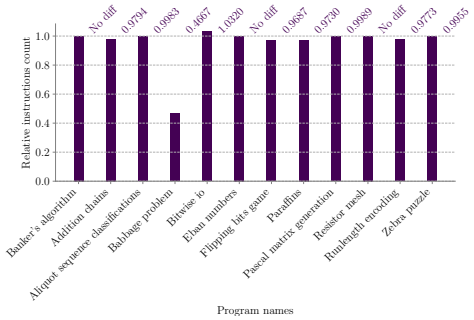


Figure 2: Vertical bars show the relative binary size in # of instructions. The smaller, the better.

4.1 Methodology

To evaluate our superoptimization pipeline, we run it on each program with four Souper configurations:

(1) Inferring only replacements for constant values
(2) Inferring replacements with no more than 2 instructions, i.e. a new replacement is composed by no more than two instructions
(3) CEGIS (Counter Example Guided Inductive Synthesis, algorithm developed by Gulwani et al. [9])
(4) Enumerative synthesis with no replacement size limit

In the rest of the paper, we report on the best configuration per program. Our appendix website contains the results for all configurations and all programs.

With respect to correctness, we rely on Souper's verification to check that every replacement on each program is correct. That means that the superoptimized programs are semantically equivalent. Every candidate search is done with a 300 seconds timeout. For each program, we report the best optimized case over all mentioned configurations. To discuss the results, we report the relative instruction count before and after superoptimization.

For the baseline program, we ask LLVM to generate WASM programs based on the 'wasm32-unknown-unknown' target with the -O3 optimization level. Our experiments run on an Azure machine with 8 cores (16 virtual CPUs) at 3.20GHz and 64GB of RAM.

4.2 Results

Figure 2 shows the relative size improvement with superoptimization. The median size reduction is 0.33% of the original instruction count over the tested programs. From the 12 tested programs, 8 have been improved using our pipeline whereas 3 have no changes and 1 is bigger (**Bitwise IO**). The most superoptimized program is **Babbage problem**, for which the resulting code after superoptimization is 46.67% smaller than the baseline version.

We now discuss the **Babbage problem** program, originally written in 15 lines of C code[3]. The pipeline found 3 successful code replacements for superoptimization out of 7 candidates. The best

[3] http://www.rosettacode.org/wiki/Babbage_problem#C

superoptimized version contains 21 instructions, which is much less than the original which has 45 instructions. The superoptimization code difference program is shown in Figure 3. Our pipeline, using Souper, finds that the loop inside the program can be replaced with a **`const`** value in the top of the stack, see lines 8 and 12 in Figure 3. The value, 25264, is the solution to the Babbage problem. In other terms, the superoptimization pipeline has successfully symbolically executed the problem.

The **Babbage problem** code is composed of a loop which stops when it discovers the smaller number that fits with the Babbage condition below.

```
while((n * n) % 1000000 != 269696) n++;
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with llvm-opt. However, llvm-opt cannot unroll a **`while`**-loop because the loop count is not known at compile time. Additionally, this is a specific optimization that does not generalize well when optimizing for code size and requires a significant amount of time per loop.

On the other hand, Souper can deal with this case. The variable that fits the Babbage condition is inferred and verified in the SMT solver. Therefore the condition in the loop will always be false, resulting in dead code that can be removed in the final stage that generates WASM from bitcode.

In the case of the **Bitwise IO** program, we observe an increase in the number of instructions after superoptimization. From the original number of 875 instructions, the resulting count after the Souper pass is increased to 903 instructions. In this case, Souper finds 4 successful replacements out of 207 possible ones. Looking at the changes, it turns out that the LLVM IR code costs less than the original following the Souper cost function. However, the WebAssembly LLVM backend (wasm-ld tool) that transforms LLVM to WASM creates a longer WASM version. This a consequence of the discussion on Souper in subsection 2.3. In practice, it is straightforward to detect and discard those cases.

```
1    (func $__original_main (type 2) (result i32)
2  -   (local i32 i32 i32 i32)
3  +   (local i32)
4      global.get 0
5      i32.const 16
6      i32.sub
7      local.tee 0
8  -   {...Removed code}
9      global.set 0
10     local.get 0
11 -   local.get 1
12 +   i32.const 25264
13     i32.store
14     i32.const 1024
15     local.get 0
16     call $printf
17     drop
18     local.get 0
19     i32.const 16
20     i32.add
21     global.set 0
22     i32.const 0)
```

```
1  -    i32.const -1
2  -    local.set 1
3  -    block  ;; label = @1
4  -      loop  ;; label = @2
5  -        local.get 1
6  -        i32.const 1
7  -        i32.add
8  -        local.tee 1
9  -        local.get 1
10 -        i32.mul
11 -        local.tee 2
12 -        i32.const 1000000
13 -        i32.rem_u
14 -        local.set 3
15 -        local.get 2
16 -        i32.const 2147483647
17 -        i32.eq
18 -        br_if 1 (;@1;)
19 -        local.get 3
20 -        i32.const 269696
21 -        i32.ne
22 -        br_if 0 (;@2;)
23 -      end
24 -    end
```

Figure 3: Output of superoptimization WASM bytecode for the Babbage problem program.

4.3 Correctness Checking

To validate the correctness of the superoptimized program we perform a comparison of the output of the non-superoptimized program and the superoptimized one. For 7/12 programs, both versions, non-superoptimized and superoptimized, behave equally and return the expected output. For 5/12 programs we cannot run them because the code generated for the target WASM architecture lacks required runtime primitives.

5 RELATED WORK

Our work spans the areas of compilation, transformation, optimization and web programming. Here we discuss three of the most relevant works that investigate superoptimization and web technologies.

Churchill et al. [4] use STOKE [1] to superoptimize loops in large programs such as the Google Native Client [3]. They use a bounded verifier to make sure that every generated optimization goes through all the checks for semantic equivalence. We apply the concept of superoptimization to the same context, but with a different stack, WebAssembly. Also, our work offloads the problem of semantic checking to an SMT solver, included in the Souper internals.

Emscripten is an open source tool for compiling C/C++ to the Web Context. Emscripten provides both, the WASM program and the JavaScript glue code. It uses LLVM to create WASM but it provides support for faster linking to the object files. Instead of all the IR being compiled by LLVM, the object file is pre-linked with WASM, which is faster. The last version of Emscripten also uses the WASM LLVM backend as the target for the input code.

To our knowledge, at the time of writing, the closest related work is the "souperify" pass of Binaryen [14]. It is implemented as an additional analysis on top of the existing ones. Compared to our pipeline, Binaryen does not synthesize WASM code from the Souper output.

6 CONCLUSION

We propose a pipeline for superoptimizing WebAssembly. It is a principled integration of two existing tools, LLVM and Souper, that provides equivalent and smaller WASM programs.

We have shown that the superoptimization pipeline works on a benchmark of 12 WASM programs. As for other binary formats, superoptimization of WebAssembly can be seen as complementary to standard optimization techniques. Our future work will focus on extending the pipeline to source languages that are not handled, such as TypeScript and WebAssembly itself.

ACKNOWLEDGEMENT

This work has been partially supported by WASP program and by the TrustFull project financed by the Swedish Foundation for Strategic Research. We thank John Regehr and the Souper team for their support.

REFERENCES

[1] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[2] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. 2016. Learning to superoptimize programs. *arXiv e-prints* 1, 1, Article arXiv:1611.01787 (Nov. 2016), 10 pages. arXiv:cs.LG/1611.01787

[3] Google Chrome. 2013. Welcome to Native Client - Google Chrome. Retrieved Dec 27, 2019 from https://developer.chrome.com/native-client

[4] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. *SIGPLAN Not.* 52, 4 (April 2017), 313–326. https://doi.org/10.1145/3093336.3037754

[5] Emscripten Community. 2015. emscripten-core/emscripten. Retrieved 2019-12-11 from https://github.com/emscripten-core/emscripten

[6] LLVM community. 2019. LLVM 10 documentation. Retrieved 2019-12-12 from http://llvm.org/docs/

[7] World Wide Web Consortium. 2016. WebAssembly becomes a W3C Recommendation. Retrieved Dec 5, 2019 from https://www.w3.org/2019/12/pressrelease-wasm-rec.html

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[9] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73. https://doi.org/10.1145/1993316.1993506

[10] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/3357390.3361029

[11] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. https://doi.org/10.1145/3140587.3062363

[12] Massalin Henry. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (Nov 1987), 122–126. https://doi.org/10.1145/36177.36194

[13] LLVM. 2019. WebAssembly lld port — lld 10 documentation. https://lld.llvm.org/WebAssembly.html

[14] WebAssembly. Development of WebAssembly and associated infrastructure. 2017. emscripten-core/emscripten. Retrieved 2019-12-11 from https://github.com/WebAssembly/binaryen

[15] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv e-prints* 2, 1, Article arXiv:1711.04422 (Nov. 2017), 10 pages. arXiv:cs.PL/1711.04422

[16] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings ASPLOS'13*. ACM, New York, NY, USA, 305–316. event-place: Houston, Texas, USA.

CROW: CODE DIVERSIFICATION FOR WEBASSEMBLY

Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
NDSS 2021, MADWeb

CROW: Code Diversification for WebAssembly

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Orestis Floros
KTH Royal Institute of Technology
Stockholm, Sweden
forestis@kth.se

Oscar Luis Vera Perez
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
oscar.vera-perez@inria.fr

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

***Abstract*—The adoption of WebAssembly increases rapidly, as it provides a fast and safe model for program execution in the browser. However, WebAssembly is not exempt from vulnerabilities that can be exploited by malicious observers. Code diversification can mitigate some of these attacks. In this paper, we present the first fully automated workflow for the diversification of WebAssembly binaries. We present CROW, an open-source tool implementing this workflow through enumerative synthesis of diverse code snippets expressed in the LLVM intermediate representation. We evaluate CROW's capabilities on 303 C programs and study its use on a real-life security-sensitive program: libsodium, a modern cryptographic library. Overall, CROW is able to generate diverse variants for 239 out of $303\,(79\%)$ small programs. Furthermore, our experiments show that our approach and tool is able to successfully diversify off-the-shelf cryptographic software (libsodium).**

I. Introduction

WebAssembly is the fourth official language of the Web [36]. The language provides low-level constructs enabling efficient execution times, much closer to native code than JavaScript. It constitutes a fast and safe platform to execute programs in the browser and embedded environments [21]. Consequently, the adoption of WebAssembly has been rapidly growing since its introduction in 2015. Nowadays, languages such as Rust and C/C++ can be compiled to WebAssembly using mature toolchains and can be executed in all notable browsers.

The WebAssembly execution model is designed to be secure and to prevent many memory and control flow attacks. Still, as its official documentation admits [11], WebAssembly is not exempt from vulnerabilities that could be exploited [30]. Code diversification [5], [28] is one additional protection that can harden the WebAssembly stack. This consists in synthesizing different variants of an original program that provide the same functionalities but exhibit different execution traces. In this paper, we investigate the feasibility of diversifying WebAssembly code, which is, to the best of our knowledge, an unresearched area.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/madweb.2021.23xxx
www.ndss-symposium.org

Our contribution is a workflow and a tool, called CROW, for automatic diversification of WebAssembly programs. It takes as input a C/C++ program and produces a set of diverse WebAssembly binaries as output. The workflow is based on enumerative code synthesis. First, CROW lists blocks that are potentially relevant for diversification, second, CROW enumerates alternative instruction sequences, and third, CROW checks that the new instruction sequences are functionally equivalent to the original block. CROW builds on the idea of superdiversification [25] and extends the concept to the enumeration of a set of variants instead of synthesizing only one solution. We also take into account the specificities of WebAssembly and the details of its execution.

We evaluate the diversification capabilities of CROW in two ways. First, we diversify 303 small C programs compiled to WebAssembly. Second, we run CROW to diversify a real-life cryptographic library that natively supports WebAssembly. In both cases, we measure the diversity among binary code variants, as well as the diversity of execution traces. When measuring the diversity in binary code, we compare the WebAssembly and the machine code variants. This way we assess the ability of CROW at synthesizing variations in WebAssembly, as well as the extent to which these variations are preserved when compiling WebAssembly to machine code. Our original experiments demonstrate the feasibility of diversifying WebAssembly code. CROW generates diverse variants for $239/303\,(79\%)$ C programs. TurboFan, the optimizing compiler used in the V8 engine, preserves 99.48% of these variants. CROW successfully synthesizes variants for the cryptographic library. The variants indeed yield either different execution traces. This is promising milestone in getting a more secure Web environment through diversification.

To sum up, our contributions are:

- CROW: the first automated workflow and tool to diversify WebAssembly programs, it generates many diverse WebAssembly binaries from a single input program.
- A quantitative evaluation over 303 programs showing the capability of CROW to diversify WebAssembly binaries and measuring the impact of diversification on execution traces.
- A feasibility study of the diversification on a real-world WebAssembly program, demonstrating that CROW can handle libsodium, a state-of-the-art cryptographic library.

II. Background

A. WebAssembly

WebAssembly is a binary instruction format for a stack-based virtual machine. It is designed to address the problem of safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware or languages and can run in a standalone Virtual Machine (VM) or in other environments such as Arduino [20]. A paper by Haas et al. [21] formalizes the language and its type system, and explains the design rationale.

Listing 1 and 2 illustrate WebAssembly. Listing 1 presents the C code of two functions and Listing 2 shows the result of compiling these two functions into a WebAssembly module. The `type` directives at the top of the module declare the function: the types of its parameters and the type of the result. Then, the definitions for the function follow. These definitions are sequences of stack machine instructions. At the end, the `main` function is exported so that it can be called from outside this WebAssembly module, typically from JavaScript. WebAssembly has four primitive types: integers (`i32` and `i64`) and floats (`f32` and `f64`) and it includes structured instructions such as `block`, `loop` and `if`.

Listing 1: C function that calculates the quantity $2x + x$

```
int f(int x) { return 2 * x + x; }

int main(void) { return f(10); }
```

Listing 2: WebAssembly code for Listing 1.

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;1;) (func (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
  (func (;1;) (type 1) (result i32)
    i32.const 10
    call 0)
  (export "main" (func 1)))
```

WebAssembly is characterized by an extensive security model [11] founded on a sandboxed execution environment that provides protection against common security issues such as data corruption, code injection and return oriented programming (ROP). However, WebAssembly is no silver bullet and is vulnerable under certain conditions [30]. This motivates our work on software diversification as one possible mitigation among the wide range of security counter-measures.

B. Motivation for Moving Target Defense in the Web

The distribution model for web computing is as follows: build one binary and distribute millions of copies, all over the world, which run on browsers. In this model an attacker has two key advantages over the developers: she has a runtime environment that she fully controls and observes in any possible way. Consequently, when she finds a flaw in this virtually transparent environment, knowing that this flaw is present in the millions of copies that have been distributed over the world, she can exploit the flaw at scale.

The developers can never assume that they can control the web browser. Yet, they can challenge the second advantage of the attacker, known as the break-once-break-everywhere advantage. The developers can stop distributing clones of the binary and distribute diverse versions instead, as suggested by the pioneering software diversification works of Cohen [12] and Forrest et al. [19].

In the context of diversification, moving target defense [40] means distributing diverse variants constantly. In the context of the web, it means distributing a different variant at each HTTP request. Moving target defense is appropriate for mitigating yet unknown vulnerabilities. The diversification technique does not always remove the potential flaws, yet the vulnerabilities in the diversified binaries can be located in different places. With moving target defense, a successful attack on one browser cannot be performed on another browser with the same effectiveness. The diversified binaries that CROW outputs can be used interchangeably over the network, in a moving target defence choreographed over the web.

To sum up, by combining moving target defense deployment to diversification, we reduce the information asymmetry between the Web attacker and the defender, increasing the uncertainty and complexity of successful attacks over all client browsers [16], [42].

III. CROW's Diversification Technique

In this section we describe the workflow of CROW for diversifying WebAssembly programs. First we introduce the main concepts behind CROW. Then, we describe each stage of the workflow and we discuss the key implementation details.

A. Definitions

In this subsection we define the key concepts for CROW.

Definition 1: Block (based on Aho et al. [2]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.

Definition 2: Program state (based on Mangpo et al. [35]): At any point in time, the program state S is defined as the collection of local and global variables, and, the program counter pointing to the next instruction.

Definition 3: Pure block: A block B is said to be pure if and only if, given the program state S_i, every execution of B produces the same state S_o.

Definition 4: Functional equivalence modulo program state (based on Le et al. [29]): Let B_1 and B_2 be two blocks. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.

Definition 5: Code replacement: Let P be a program and T a pair of blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both pure as defined in Definition 3 and functionally equivalent as defined in Definition 4. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.

CROW generates new program variants by finding and applying code replacements as defined in Definition 5. A program variant could be produced by applying more than one candidate code replacement. For example, the tuple, composed by the code blocks in Listing 3 and Listing 4, is a code replacement for Listing 2.

Listing 3: WebAssembly pure code block from Listing 2.

```
local.get 0
i32.const 2
i32.mul ; 2 * x ;
```

Listing 4: Code block that is functionally equivalent to Listing 3

```
local.get 0
i32.const 1
i32.shl ; x << 1 ;
```

B. Overview

CROW synthesizes variants for WebAssembly programs. We assume that the programs are generated through the LLVM compilation pipeline. This assumption is motivated as follows: first, LLVM-based compilers are the most popular compilers to build WebAssembly programs [30]; second, the availability of source code (typically C/C++ for WebAssembly) provides a structure to perform code analysis and produce code replacements that is richer than the binary code.

CROW takes as input a C/C++ program and produces a set of unique, diversified WebAssembly binaries. Figure 1 shows the stages of this workflow. The workflow starts with compiling the input program into LLVM bitcode using clang. Then, CROW analyzes the bitcode to identify all pure blocks and to synthesize a set of candidate replacements for each pure block. This is what we call the *exploration* stage. In the *generation* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants. Finally, those bitcode variants are compiled to WebAssembly binaries that can be sent to web browsers.

Challenges. The concept of diversifying WebAssembly programs is novel and it is arguably hard for the following reasons. First, WebAssembly is a structured binary format, without goto-like instructions. This prevents the direct application of a wide range of diversification operators based on goto [41]. Second, the existing transformation and diversification tools target instruction sets larger than the one of WebAssembly [39]. This limits the efficiency of diversification, and the possibility of searching for a large number of equivalent code replacements. We address the former challenge using the LLVM intermediate representation as the target for diversification. We address the latter challenge by tailoring a superoptimizer for LLVM, using its subset of the LLVM intermediate representation. In particular, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly (for example, `freeze` instructions), which is an essential step to get executable diversified WebAssembly code.

C. Exploration stage

Given a program P for which we want to generate WebAssembly variants, the exploration stage of CROW identifies all pure blocks in the LLVM bitcode of P. CROW considers every directed acyclic graph contained in one function as a pure block. Then, CROW searches for code replacements for each one of them.

The generation of a code replacement consists of two steps. First, the synthesis of the new block, and, second, equivalence checking. Every variant block that passes the equivalence check is stored for use in diversification. The synthesis of block variants consists of enumerating all possible blocks that can be built as a combination of a given number of instructions, bounded by a maximum value to keep a tractable synthesis space.

There are two parameters to control the size of the search space and hence the time required to traverse it. On one hand, one can limit the size of the variants. In our experiments we limit the block variants to a maximum of 50 instructions. On the other hand, one can limit the set of instructions that are used for the synthesis. In our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer). This configuration allows the user to find a trade-off between the amount of variants that are synthesized and the time taken to produce them.

Listing 5: Listing 1 in LLVM's intermediate representation.

```
define i32 @f(i32) {
 %2 = mul nsw i32 %0,2
 %3 = add nsw i32 %0,%2

 ret i32 %3
}

define i32 @main() {
 %1 = tail call i32 @f(i32 10)
 ret i32 %1
}
```

Block A

```
%2 = mul nsw i32 %0,2
```

Block B

```
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2
```

In Listing 5 we illustrate the LLVM bitcode representation of Listing 1. In this bitcode, CROW identifies two pure blocks in function `f()`, which are displayed on the right part of the listing, in gray and green. The first pure block is composed of one single instruction (line 2) that performs the `2*x` multiplication. The second block has two instructions, one multiplication and one addition.

Using CROW, it is possible to diversify both blocks. For example, using a maximum of 1 instruction per replacement and searching over the complete bitcode instruction set, a potential replacement for Block A is: `%2 = shl nsw i32 %0,1 %`. This replacement calculates the same expression `2*x`, using a shift left operation.

To determine the equivalence between a pure block and a candidate replacement, we use an equivalence checker based on SMT [17]. In our example, the checker would prove that there cannot be a value of x such that $2 * x \neq x \ll 1$. In general, if no such counter-example exists, then the functional equivalence is assumed. On the other hand, if there exists an input resulting in different outputs for a block and a variant, then they are proven not equivalent and the variant is discarded.

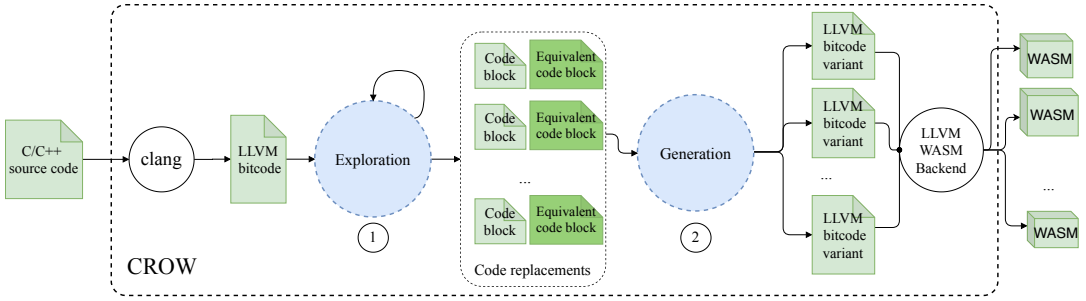


Fig. 1: CROW's workflow for diversifying WebAssembly programs.

D. Generation stage

In this stage, we select and combine code replacements that have been synthesized during the exploration stage, in order to generate WebAssembly binary variants. We apply each code replacement to the original program to produce a LLVM IR variant. Then, this IR is compiled into a WebAssembly binary. CROW generates WebAssembly binaries from all possible combinations of code replacements as the power set over all code replacements.

After the exploration phase, it is possible that two subsets of code replacements overlap, *i.e.,* they produce the same WebAssembly binary. The overlap between blocks is explained as follows: Let $S = \{(B_1, R_1), (B_1, R_2), \cdots, (B_n, R_m)\}$ be a set of candidate replacements over a program P. If two blocks from the original program B_i, B_j, $j \neq i$, overlap, *i.e.,* the intersection of $\mathrm{CFG}(B_i)$[1] and $\mathrm{CFG}(B_j)$ is not empty, then only the replacements of the largest original block are preserved when combining blocks.

In this example, the exploration stage synthesizes $6 + 1$ bitcode variants for the considered blocks respectively, which results in 14 module variants (the power set combination size). Yet, the generation stage would eventually generate 7 variants from the original WebAssembly binary. This gap between the number of potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composing several variants into one.

E. Implementation

The majority of the WebAssembly applications are built from C/C++ source code using the LLVM toolchain. Consequently, the implementation of CROW is based on LVVM/ Furthermore, CROW extends Souper [38], a superoptimizer for LLVM that aims to reduce the size of binary code. Souper has its own intermediate representation, which is a subset of the LLVM IR.

To extract code blocks, we scan LLVM modules, looking for instructions that return integer-typed values. Each such instruction is considered as the exit of a code block. Souper's representation of a code block is built as a backward traversal process through the dependencies of the detected instruction. If memory loads or function calls are found, the backward traversal process is stopped and the current instruction is considered as an input variable for the code block. Notice that, by construction, Souper's translation is oblivious to the memory model, thus, it cannot infer string data types or other abstract data types. The translation from Souper IR to a BitVector SMT theory is done on the fly. Souper uses the z3[2] solver to check the equivalence between a code block original and a potential replacement for it.

We now summarize the main changes that we implement in Souper and in the LLVM backend in order to support diversification. Souper, as a superoptimizer, aims at generating a single variant that is smaller than the original, yet we want to obtain as many blocks as possible. To achieve automatic diversification, we modify Souper to disable the key cost restriction functions, data-flow pruning and peephole optimizations, all being detrimental for diversification. In order to increase the number of variants that CROW can generate, CROW parallelizes the process of replacement synthesis.

In addition, CROW orchestrates a series of Souper executions with various configurations (in particular the size of the replaced expression). Finally, we carefully fine-tune a set of 19 Souper options to ensure that the search is effective for diversification in feasible time.

In the generation stage of CROW, we also modify Souper to amplify the generation of WebAssembly binary diversity. Initially, Souper generates a single bitcode variant, inserting all replacements at once. We modify it so that we can obtain a combination of code replacements. Finally, on the LLVM side, we disable all peephole optimizations in the WebAssembly backend, in particular instructions merging and constant folding. This aims to preserve the variations introduced in the LLVM bitcode during the generation of binaries.

The implementation of CROW is publicly available for sake of open science and can be reviewed at https://github.com/KTH/slumps/tree/master/crow.

IV. Evaluation Protocol

To evaluate the capabilities of CROW to diversify WebAssembly programs, we formulate the following research

[1] $\mathrm{CFG}(A)$ refers to backward Control Flow Graph starting at inst. A.

[2] https://github.com/Z3Prover/z3

questions:

RQ1: **To what extent are the program variants generated by CROW statically different?** We check whether the WebAssembly binary variants produced by CROW are different from the original WebAssembly binary. Then, we assess whether the generation of x86 machine code performed by V8's WebAssembly engine preserves CROW's transformations.

RQ2: **To what extent are the program variants generated by CROW dynamically different?** It is known that not all diversified programs produce distinguishable executions [15], sometimes it is impossible to observe different behaviors between variants. We check for the presence of different behaviors with a custom WebAssembly interpreter, characterizing the behavior of a WebAssembly program by its stack operation trace.

RQ3: **To what extent can CROW be applied to diversify real-world security-sensitive software?** We assess the ability of CROW to diversify a state-of-the-art cryptographic library for WebAssembly, libsodium [18].

A. Corpus

We answer RQ1 and RQ2 with a corpus of programs appropriate for our experiments. We take programs from the Rosetta Code project[3]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains a wide range of tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. Next, we apply a number of filters. We discard 1) all programs that do not compile with `clang`, 2) all interactive programs requiring input from users *i.e.,* invoking functions like `scanf`, 3) all programs that contain more than 100 blocks, 4) all programs without termination, 5) all programs with non-deterministic operations, for example, programs working with time or random functions. This filter produces a final set of 303 programs.

The result is a corpus of 303 C programs. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

B. Protocol for RQ1

With RQ1, we assess the ability of CROW to generate WebAssembly binaries that are different from the original program. For this, we compute a distance metric between the original WebAssembly binary and each binary generated by CROW. Since WebAssembly binaries are further transformed into machine code before they execute, we also check that this additional transformation preserves the difference introduces by CROW in the WebAssembly binary. We use the Turbofan ahead-of-time compiler of V8, with all its possible optimizations, to generate a x86 binary for each WebAssembly binary. Then, we compare the x86 version of each variant against the x86 binary corresponding to the original WebAssembly binary.

We compare the WebAssembly and machine code of each program and its variant using Dynamic Time Warping (DTW) [31]. DTW computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is actually a distance metric, called DTW. The larger the DTW distance, the more different the two sequences are. In our case, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain two DTW distance values for each program-variant pair: one at the level of WebAssembly code and the another one at the level of x86 code. Metric 1 below defines these metrics.

Metric 1: dt_static: Given two programs P_X and V_X written in X code, $dt\_static(P_X, V_X)$, computes the DTW distance between the corresponding program instructions for representation X ($X \in \{Wasm, x86\}$). A $dt\_static(P_X, V_X)$ of 0 means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.

We run CROW on our corpus of 303 programs. We configure CROW to run with a diversification timeout of 6 hours per program. For each program, we collect the set of generated variants. For all pairs program, variant that are different, we compute both dt_static for WebAssembly and x86 representations.

The key property we consider is as follows: if $dt\_static(P_{Wasm}, P'_{Wasm}) > 0$ and $dt\_static(P_{x86}, P'_{x86}) > 0$, this means that both programs are still different when compiled to machine code, and we conclude that V8's compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program (including the original).

C. Protocol for RQ2

For RQ2, we compare the executions of a program and its variants for a given input. In this experiment, we characterize the execution of a WebAssembly binary according to its trace of stack operations.

This method of tracing allows us to evaluate CROW's effect on program execution according to the WebAssembly specification, independently of any specific engine.

For each execution of a WebAssembly program, we collect a trace of stack operations. These traces are composed of stack-type instructions: `push <value>` and `pop <value>`. All traces are ordered with respect to the timestamp of the events. We compare the traces of the original program against those of the variants with DTW. DTW computes the global alignment between two traces and provides a value for the cost of this alignment.

Metric 2: dt_dyn: Given a program P and a CROW generated variant P', dt_dyn(P,P'), computes the DTW distance between the corresponding stack operation traces collected during their execution. A dt_dyn of 0 means that both traces are identical. The higher the value, the more different the stack operation traces.

To answer RQ2 we compute Metric 2 for a study subject program and all the unique program variants generated by CROW in a pairwise comparison. The pairwise comparison

[3]http://www.rosettacode.org/wiki/Rosetta_Code

allows us to compare the diversity between variants as well. We use SWAM[4] to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [10].

The builtin WebAssembly API for JavaScript is usually mutable, thus, the same model for traces collection can be implemented on top of V8. In other words, a custom interpreter can be implemented in order to collect the traces in the browser or standalone JavaScript engines. This validates the usage of SWAM to study the traces diversity.

D. Protocol for RQ3

In RQ3, we assess the ability of CROW to diversify a mature and complex software library related to security. We choose the libsodium [18] cryptographic library, which natively compiles to WebAssembly. With 3752 commits contributed by 96 developers, its API provides the basic blocks for encryption, decryption, signatures and password hashing. We experiment with code revision 2b5f8f2b, which contains 45232 lines of C code. Libsodium has 102 separate WebAssembly modules that we use as input for CROW. Each module corresponds to one C file that encompasses a set of related functions.

To answer RQ3, we run CROW on the libsodium bitcodes, generating a set of WebAssembly variants. Then, we assess both binary code diversity and behavioural diversity between the variants and the original libsodium, using the same techniques as in RQ1 and RQ2.

Collecting traces The libsodium repository includes an extensive test suite of 77 tests, where one test is one usage scenario. We use this test suite to measure the trace diversity among program variants. Since some test traces are larger than 1 GB each, we focus on reasonably sized tests: we select the $41/77$ test cases that produce a trace containing less than 50 million events each.

To measure the relative trace diversification for each test, we normalize the dt_dyn used in RQ2 by dividing it with the length of the original trace. This allows us to compare the relative success of CROW's diversification technique across different tests.

Since libsodium uses a pseudo-number generator, we set a static seed when executing libsodium, so that the diversity observed in traces is only due to CROW's diversification. This seed is given to the `arc4random` API used by libsodium in WebAssembly. To quantify the effectiveness of our diversification technique, we compare the trace distance produced by our technique with the trace distance that occurs when the seed is changed (baseline).

V. Experimental Results

In this section we present the results for the research questions formulated in section IV.

[4]https://github.com/satabin/swam

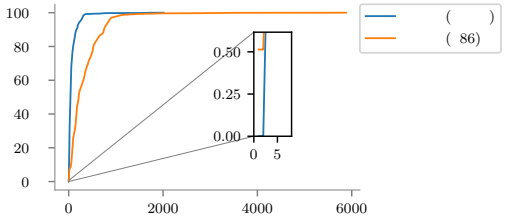


Fig. 2: Cumulative distribution for all pairwise comparisons between a program and its variants. Each line corresponds to a different program representation.

A. To what extent are the program variants generated by CROW statically different?

We run CROW on 303 C programs compiled to WebAssembly. CROW produces at least one unique program variant for $239/303$ programs. For the rest of the programs ($64/303$), the timeout is reached before CROW can find any valid variant.

We subsequently perform a manual analysis of the programs that yield more than 100 unique WebAssembly variants. This reveals one key reason that favors a large number of unique WebAssembly variants: the programs include bounded loops. In these cases CROW synthesizes variants for the loops by unrolling them. Every time a loop is unrolled, the loop body is copied and moved as part of the outer scope of the loop. This creates a new, statically different, program. The number of programs grows exponentially with nested loops.

A second key factor for the synthesis of many variants relates to the presence of arithmetic. Souper, the synthesis engine used by CROW, is effective in replacing arithmetic instructions by equivalent instructions that lead to the same result. For example, CROW generates unique variants by replacing multiplications with additions or shift left instructions (Listing 8). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 9).

Listing 8: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 9: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

We now discuss the prevalence of the transformations made by CROW when the WebAssembly binaries are transformed to machine code, specifically with the V8's engine. In Figure 2 we plot the cumulative distribution of dt_static, comparing WebAssembly binaries (in blue) and x86 binaries (in orange). The figure plots a total of 103003 dt_static values for each representation, two values for each variant pair comparison (including original) for the 239 program. The value on the y-axis shows which percentage of the total comparisons lie

Listing 10: Excerpt of WebAssembly program p74: CROW replaces a loop by a constant.

```
local.set 1
loop ;; label = @1
  ...
end
...
i32.store
```

```
local.get 0
i32.const 25264
i32.store
```

below the corresponding dt_static value on the x-axis. Since we measure the distances between original programs and WebAssembly variants, then 100% of these binaries have $dt\_static > 0$. Let us consider the x86 variants: dt_static is strictly positive for 99.48% of variants. In all these cases, the V8 compilation phase does not undo the CROW diversification transformations. Also, we see that there is a gap between both distributions, the main reason is the natural inflation of machine code. For example, two variants that differ by one single instruction in WebAssembly, can be translated to machine code where the difference is increased by more than one machine code instruction.

The zoomed subplot focuses on the beginning of the distribution, it shows that the dt_static is zero for 0.52% of the x86 binaries. In these cases the V8 TurboFan compiler from WebAssembly to x86 reverts the CROW transformations. We find that CROW produces at least one of these reversible transformations for $34/239$ programs. Listing 11 shows one of the most common transformations that is reversed by TurboFan, according to our experiments.

Listing 11: Replacement in WebAssembly that is translated to the same x86 code by V8-TurboFan.

```
i32.const -<n>
i32.sub
```

```
i32.const <n>
i32.add
```

We look at the cases that yield a small number of variants. There is no direct correlation between the number of identified blocks and the number of unique variants. We manually analyze programs that include a significant number of pure blocks, for which CROW generates few variants. We identify two main challenges for diversification.

1) Constant computation We have observed that Souper searches for a constant replacement for more than 45% of the blocks of each program while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because CROW is oblivious to a memory model.

2) Combination computation The overlap between code replacements, discussed in subsection III-D, is a second factor that limits the number of unique variants. CROW can generate a high number of variants, but not all replacement combinations are necessarily unique.

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the 239 programs with their variants. The ratio of size change between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for all Rosetta programs. This limited impact on the binary size of the variants is good news because they are meant to be distributed to browsers over the network.

Answer to RQ1

CROW is able to generate diverse variants of WebAssembly programs for $239/303\,(79\%)$ programs in our corpus. We observe that programs that include bounded loops and arithmetic expressions are highly prone to diversification. V8's TurboFan compilation to x86 code preserves 99.48% of the transformations performed by CROW. To our knowledge, this is the first ever realization of automated diversification for WebAssembly.

B. To what extent are the program variants generated by CROW dynamically different?

Now, we focus on the 41 programs that have at least 9 unique WebAssembly variants in order to study the diversity of execution traces. We apply the protocol described in subsection IV-C by executing the WebAssembly programs and their unique variants in order to collect the stack operation traces. Then, we compare the traces of each pair of original program and a variant. We run 1906 program executions and we perform 98774 trace pair comparisons.

Table I summarizes the observed trace diversity, as captured by dt_dyn (Metric 2), among each program and their variants. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. The table summarizes the distribution of distances between stack operation trace pairs: the minimum value, the maximum value, the median value, the percentage of values equal to zero and the percentage of values greater than zero. The programs are sorted with respect to the number of unique variants. The green highlight color in $> 0\%$ columns represents more than 50% of non-zero comparisons, *i.e.,* high diversification. For instance, the first row shows the trace diversity for `p96`, where 99.70% of the pairwise comparisons between all collected traces have a different dt_dyn.

For the stack operation traces, all programs have at least one variant that produces a trace different from the original. All but one (`p81`) programs have the majority of variants producing a different stack operation trace. This shows the real effectiveness of CROW for diversifying stack operation traces.

We manually analyze variants with high and low trace diversity. We observe that constant inferring is effective at changing the stack operation trace. For instance, for program `p74` shown in Listing 10, CROW removes a loop by replacing it with a constant assignment. The execution of this variant produces traces that are different because the loop pattern is not visible anymore in the trace, and consequently, the distance between the original and the variant traces is large.

	NAME	#var	Σ	Min	Max	Median	0 %	> 0 %
1	**p96**	220	15	0	24062	820	0.30	99.70
2	p56	192	36	0	45420	1416	1.84	98.16
3	p78	159	35	0	20501	759	1.52	98.48
4	p111	144	45	0	2114	520	3.74	96.26
5	**p166**	101	152	0	44538	66	45.80	54.20
6	p122	91	34	0	46026	6434	0.24	99.76
7	p67	89	77	0	94036	85692	0.29	99.71
8	p68	85	10	0	10554	260	3.64	96.36
9	p80	78	9	0	17238	618	3.92	96.08
10	p204	77	42	0	36428	3356	0.33	99.67
11	p183	76	9	0	90628	84402	0.57	99.43
12	p136	62	70	0	62953	58028	0.60	99.40
13	p167	46	232	8	888	724	0.00	100.00
14	p226	42	13	0	90736	74476	8.26	91.74
15	p99	38	74	16	9936	5037	0.00	100.00
16	p18	36	7	0	15620	145	1.10	98.90
17	**p140**	29	17	0	13280	172	6.59	93.41
18	p59	27	6	0	85390	40	1.43	98.57
19	p199	21	87	0	27482	728	4.68	95.32
20	**p91**	21	21	0	50002	228	43.81	56.19
21	p223	21	115	16	40911	632	0.00	100.00
22	p168	20	6	0	22200	18896	2.20	97.80
23	p174	18	40	6	6566	6395	0.00	100.00
24	**p81**	17	86	0	4419	0	84.62	15.38
25	p141	17	6	8	2894	132	0.00	100.00
26	p108	16	6	0	85168	79903	8.97	91.03
27	p98	15	4	0	33	25	6.06	93.94
28	p89	14	45	10	15952	89	0.00	100.00
29	p36	14	52	312	33266	30298	0.00	100.00
30	**p135**	13	5	0	20288	20163	3.57	96.43
31	p161	12	91	240	9792	1056	0.00	100.00
32	p147	12	32	0	54071	21274	7.14	92.86
33	p11	10	38	29798	51846	35119	0.00	100.00
34	p125	10	51	0	4399	4368	7.14	92.86
35	p131	9	4	140	1454	685	0.00	100.00
36	p69	9	48	28	29243	28956	0.00	100.00
37	p134	9	20	4	514	186	0.00	100.00
38	**p74**	9	19	126	8332	6727	0.00	100.00
39	p79	9	97	4	29	16	0.00	100.00
40	p33	9	52	4	2342	15	0.00	100.00
41	p157	9	64	36	242	166	0.00	100.00

TABLE I: Dynamic diversity for 41 diversified WASM programs. The dynamic diversity is captured by dt_dyn between traces. The rows are sorted by the number of unique variants per program. The table is structured as follows: the first, second and third columns contain the program id, the number of unique variants and the overall sum of all blocks replacements respectively. Following, the stats for the dt_dyn metric. The colorized cells in the $> 0\%$ column represent high diversification.

Listing 12: Statically different WebAssembly replacements with the same behavior, gray for the original code, green for the replacement.

```
(1) i32.lt_u  i32.lt_s      (3) i32.ne        i32.lt_u
(2) i32.le_s  i32.lt_u      (4) local.get 6   local.get 4
```

We note that there is no relation between the trace distance and the number of block replacements. A high trace distance does not necessarily imply a high number of replacements. For instance, program `p135` has only 4 possible replacements overall its 5 identified blocks yet a median dt_dyn of 20163.

We subsequently analyze the cases where diversification is not reflected in stack operation traces. For example, more than 40% of the pairwise dt_dyn distances for `p166`, `p91` and `p81` are equal to zero. This indicates a lower diversity among the population of variants, than for all the other programs. This happens because some variants have two different bitcode instructions (original and replacement) that trigger the same stack operations. The instructions in Listing 12 are concrete cases of such kind of replacements. The four cases in Listing 12 leave the same value in the stack operation trace. For each case, the original instruction and the replacement are semantically equal in the program domain. The fourth case is a local variable index reallocation, this replacement only changes the index of the local variable but not the event in the stack operation trace. These replacements are sound, produce statically diverse code, but they are not useful to dynamically diversify the original program. This confirms the complementary of using static and dynamic metrics to assess diversification.

The effectiveness of CROW on diversifying stack operation traces is significant. In a security context, such diverse stack operation traces are likely to mitigate potential side-channel attacks [30]. Notably, the attacks based on code profiling are affected when the executed opcodes and the corresponding profiles are different [37].

Answer to RQ2

CROW is successful at generating diverse WebAssembly variant programs, for which we are able to observe different stack operation traces. In other words, CROW generates dynamically different binaries, and ensures that variants of a given program yield different stack operation traces.

C. To what extent can CROW be applied to diversify real-world security-sensitive software?

We run CROW on each of the 102 modules of libsodium with a 6-hour timeout. We find $45/102$ modules that do not contain any pure block, so they are not amenable to our diversification technique. CROW produces at least one valid WebAssembly module variant for 15 of the remaining 57 modules.

Module & Description	#var	#func	Diversified Functions	#calls
argon2-core Core functions for the implementation of the Argon2 key derivation (hash) function [9].	17	6	argon2_finalize argon2_free_instance argon2_initialize	0 0 0
argon2-encoding Functions for encoding and decoding (including salting) Argon2 [9] hash strings.	11	2	argon2_decode_string argon2_encode_string	0 0
blake2b-ref Reference implementation for the BLAKE2 [4] hash function.	7	11	blake2b blake2b_salt_personal blake2b_update	0 1.46E+04 2.04E+04
chacha20_ref Reference implementation of the ChaCha20 stream cipher [6].	7	5	chacha20_encrypt_bytes stream_ietf_ext_ref_xor_ic stream_ref stream_ref_xor_ic	3.51E+06 7.62E+03 1.14E+04 1.14E+05
codecs Implementations of commonly used codecs for conversions between binary formats like Base64 [26].	79	5	sodium_base642bin sodium_base64_encoded_len sodium_bin2base64 sodium_bin2hex sodium_hex2bin	0 0 0 2.57E+05 0
core_ed25519 Implementation of the Edwards-curve Digital Signature Algorithm [8].	2	19	crypto_core_ed25519_is_valid_point	0
crypto_scrypt-common Utility and low-level API functions for the scrypt key derivation (hash) function [34].	5	5	escrypt_gensalt_r	0
pbkdf2-sha256 Implementation of the Password-Based Key Derivation Function 2 (PBKDF2) [27].	14	1	escrypt_PBKDF2_SHA256	0
pwhash_scryptsalsa208sha256 High-level API for the scrypt key derivation function [34].	8	19	crypto_pwhash_scryptsalsa208sha256	0
pwhash_scryptsalsa208sha256_nosse Same as above, but does not use Streaming SIMD Extensions (SSE).	32	3	escrypt_kdf_nosse salsa20_8	0 0
randombytes Pseudorandom number generators.	1	11	randombytes_uniform	5.61E+02
salsa20_ref Contains a reference implementation of the Salsa20 stream cipher [7].	12	2	stream_ref stream_ref_xor_ic	1.14E+04 1.14E+05
scalarmult_ristretto255_ref10 Implementation of the Ristretto255 prime order elliptic curve group [22].	29	4	scalarmult_ristretto255 scalarmult_ristretto255_base scalarmult_ristretto255_scalarbytes	0 0 0
stream_chacha20 High-level API for the ChaCha20 stream cipher [8].	2	15	crypto_stream_chacha20 crypto_stream_chacha20_ietf crypto_stream_chacha20_ietf_ext crypto_stream_chacha20_ietf_ext_xor_ic crypto_stream_chacha20_ietf_xor crypto_stream_chacha20_ietf_xor_ic crypto_stream_chacha20_xor crypto_stream_chacha20_xor_ic	6.65E+02 3.19E+03 2.66E+03 1.68E+02 1.68E+02 2.32E+03 0 1.68E+02
verify Functions used to compare secrets in constant time to avoid timing attacks.	7	6	crypto_verify_16 crypto_verify_32 crypto_verify_64	2.69E+05 3.40E+03 0
Total	**256**	**114**	**40 functions**	

TABLE II: Libsodium modules with at least one variant generated by CROW. The columns on the left include the facts about each module. The first column contains the name and the functional description of the modules. The second column, *#var* (highlighted) gives the number of unique variants generated by CROW. The third column, *#func*, lists the total amount of functions in each module. The remaining columns include a list of functions that CROW has successfully diversified and the number of calls per function in the test suite.

Table II presents the key results for these 15 successfully diversified modules. The first two columns contain the name and description of the diversified module, and, the number of unique static variants. The other columns show the total number of functions inside the module, the names of the diversified functions and the number of calls to each function in the considered tests.

Generation of WebAssembly library variants from WebAssembly module variants. The successfully diversified modules can be combined to obtain a large pool of different versions of the packaged libsodium WebAssembly library. The Cartesian product of all module variants produces in theory 1.66E+15 unique libsodium variants. Yet, it is unpractical to store and execute this large number of variants. Thus, we sample the pool of possible variants to evaluate our generated variants. First, for each of the 256 modules, we rank each module variant with respect to the number of lines changed in the final WebAssembly textual format. Then, to produce the i-th library variant, we combine the i-th variant for each module of libsodium, in order to produce maximally diversified library variants first. If a module has less than i variants, we use the original, non-diversified module. According to Table II, the maximum number of unique variants for a single module is 79 (`codecs` module). Thus, we sample 79 unique libsodium variants, ordered by the amount of diversification (the first variant contains the most changes, and so on). For each variant we execute the complete test suite to validate its correctness. All test cases successfully pass for all diversified library binaries.

Dynamic evaluation of libsodium variants. We compare the dynamic behaviour of the original libsodium and the 79 library variants. Figure 3 illustrates the distribution of dt_dyn of all collected traces for each libsodium test. The dt_dyn distance is calculated between each diversified trace

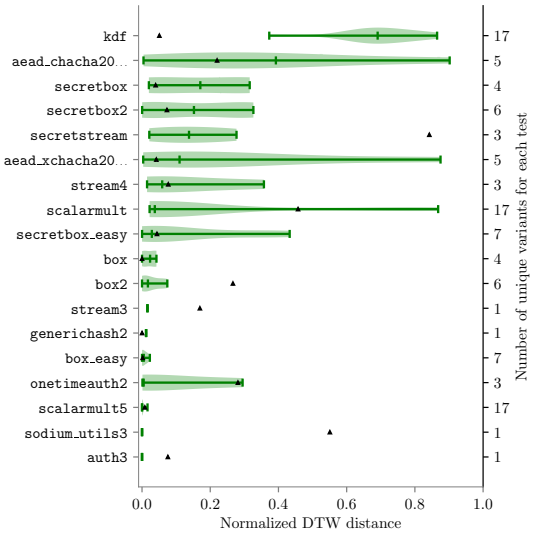


Fig. 3: Distribution of normalized dt_dyn distances over the set of libsodium variants covered by each test. The left Y axis lists the name of each test. The number of unique variants used per test is listed on the right Y axis. The black triangles point to the dt_dyn distance between two different stack operation traces of the original test with different random seeds.

and the corresponding original trace for the same test. Each horizontal bar gives the distribution of dt_dyn over the 79 diversified libraries per test. The black triangles show the dt_dyn distance between two different executions of the same test with different random seeds. They serve as a baseline to compare the artificial diversity introduced by CROW, against the natural trace diversity that appears because of random number generation.

For $18/19$ tests, we observe that CROW's diversified modules produce a different trace than the original. The wider violin plots that reach the right-hand side of the figure include variants that significantly diversify the test execution. We observe that $4/18$ tests stand out as they include variants with at least 0.8 normalized dt_dyn distance. For $6/18$ tests, there is a medium trace diversity as their dt_dyn distributions lie in the mid/left side of the plot. For the rest $8/18$ tests we observe a significantly smaller dt_dyn distance.

This means that, in the context of this cryptographic library, CROW is able to find variants that have a huge impact on the dynamic stack behaviour of the program. Meanwhile, some other replacements can have only a marginal impact during the operation of the program. One factor that can affect this is the "centrality" of the code that is being replaced. Diversified code that is called often, potentially inside loops, will have a greater impact on the stack trace of a program compared to code that is only called, for example, only during the initialization of the program.

When we compare the trace diversity against the diversity due to pseudo-number generation (black triangles in Figure 3), we observe that: for $2/18$ tests CROW trace diversification is always larger than the one due to random number generation, for $11/18$ tests there exist some variants that exhibit larger trace diversification than random number generation and for $5/18$ tests CROW trace diversification is always smaller than the one due to random number generation.

Answer to RQ3

We have successfully applied CROW to libsodium, one of the leading WebAssembly cryptography libraries. We have shown that CROW is able to create statically different variants of this real-world library, all of which being distributable to users. Our original experiments to measure the trace diversity of libsodium have proven that the generated variants exhibit significantly different execution traces compared to the original non-diversified libsodium binary. The take-away of this experiment is that CROW works on complex code.

VI. Threats to Validity

Internal: The timeout in the exploration stage is a determinant factor to generate unique variants. It is required to bound the experimental time. If the timeout is increased, the number of variants and unique variants might increase.

External: The 303 programs in our Rosetta corpus may not reflect the constructs used in the WebAssembly programs in the wild. Yet our experiment on libsodium shows that the results on the Rosetta corpus hold on real code. To increase external validity, we hope to see more benchmarks of WebAssembly programs published by the research community.

Scale: We measure behavioral diversity with DTW. We are aware that this behavioral diversity metric does not scale infinitely. To make comparisons between large execution traces, it may be necessary to use a more scalable metric. To mitigate this scale problem in future work, one option is to compare software traces using entropy analysis, as proposed by Miranskyy et al. [33].

VII. Related Work

Program diversification approaches can be applied at different stages of the development pipeline.

Static diversification: This kind of diversification consists in synthesizing, building and distributing different, functionally equivalent, binaries to end users. This aims at increasing the complexity and applicability of an attack against a large population of users [12]. Jackson et al. [24] argue that the compiler can be placed at the heart of the solution for software diversification; they propose the use of multiple semantic-preserving transformations to implement massive-scale software diversity in which each user gets their own diversified variant. Dealing with code-reuse attacks, Homescu et al. [23] propose inserting NOP instruction directly in LLVM IR to generate a variant with different code layout at each compilation. In this area, Coppens et al. [13] use compiler transformations to iteratively diversify software. The aim of their work is to prevent reverse engineering of security patches for attackers targeting vulnerable programs. Their approach, continuously applies a random

selection of predefined transformations using a binary diffing tool as feedback. A downside of their method is that attackers are, in theory, able to identify the type of transformations applied and find a way to ignore or reverse them. Our work can be extended to address this issue, providing a synthesizing solution which is more general than specific transformations.

The work closest to ours is that by Jacob et al. [25]. These authors propose the use of a "superdiversification" technique, inspired by superoptimization [32], to synthesize individualized versions of programs. In the work of Massalin, a superoptimizer aims to synthesize the shortest instruction sequence that is equivalent to the original given sequence. On the contrary, the tool developed by Jacob et al. does not output only the shortest instruction sequence, but any sequences that implement the input function. This work focuses on a specific subset of X86 instructions. Meanwhile, our approach works directly with LLVM IR, enabling it to generalize to more languages and CPU architectures. Specifically, we apply our tool on WebAssembly, something not possible with the X86-specific approach of that paper.

Runtime diversification: Previous works have attempted to generate diversified variants that are alternated during execution. It has been shown to drastically increase the number of execution traces that a side-channel attack requires to succeed. Amarilli et al. [3] are the first to propose generation of code variants against side-channel attacks. Agosta et al. [1] and Crane et al. [15] modify the LLVM toolchain to compile multiple functionally equivalent variants to randomize the control flow of software, while Couroussé et al. [14] implement an assembly-like DSL to generate equivalent code at runtime in order to increase protection against side-channel attacks. CROW focuses on static diversification of software. However, because of the specificities of code execution in the browser, this is not far from being a dynamic approach. Since WebAssembly is served at each page refreshment, every time a user asks for a WebAssembly binary, she can be served a different variant provided by CROW.

VIII. Conclusion

Security has been a major driver for the design of WebAssembly. Diversification is one additional protection mechanism that has been not yet realized for it. In this paper, we have presented CROW, the first code diversification approach for WebAssembly. We have shown that CROW is able to generate variants for a large variety of programs, including a real-world cryptographic library. Our original experiments have comprehensively assessed the generated diversity: we have shown that CROW generates diversity both among the binary code variants as well as in the execution traces collected when executing the variants. Also, we have successfully observed diverse execution traces for the considered cryptographic library, which can protect it against a range of side channel attacks.

Future work includes increasing the number of unique variants that are generated, by working on block replacement overlapping detection. Also, the exploration stage and the identification of code replacements is a highly parallelizable process, this would increase diversification performance in order to meet the demands of the internet scale.

References

[1] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale, "The MEET approach: Securing cryptographic embedded software against side channel attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1320–1333, 2015.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ch. 1, pp. 28–31.

[3] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall, "Can code polymorphism limit information leakage?" in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2011, pp. 1–21.

[4] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[5] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–26, 2015.

[6] D. J. Bernstein, "The ChaCha family of stream ciphers," 2008. [Online]. Available: http://cr.yp.to/chacha.html

[7] ——, "The Salsa20 family of stream ciphers," in *New stream cipher designs*. Springer, 2008, pp. 84–97.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[9] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 292–302.

[10] J. Cabrera Arteaga, M. Monperrus, and B. Baudry, "Scalable comparison of javascript V8 bytecode traces," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–31.

[11] D. Chen and W3C group, "WebAssembly documentation: Security," W3C, Accessed: 18 June 2020. [Online]. Available: https://webassembly.org/docs/security/

[12] F. B. Cohen, "Operating system protection through program evolution." *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[13] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–26, 2013.

[14] D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *IFIP International Conference on Information Security Theory and Practice*. Springer, 2016, pp. 136–152.

[15] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[16] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving target defense*. Springer, 2011, pp. 99–108.

[17] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[18] F. Denis, "The Sodium cryptography library," Jun 2013. [Online]. Available: https://download.libsodium.org/doc/

[19] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.

[20] R. Gurdeep Singh and C. Scholliers, "WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2019, 2019, pp. 27–36.

[21] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 185–200.

[22] M. Hamburg, H. de Valance, I. Lovecruft, and T. Arcieri, "The ristretto group," 2017.

[23] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–11.

[24] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*. Springer, 2011, pp. 77–98.

[25] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *International Workshop on Security*. Springer, 2008, pp. 100–120.

[26] S. Josefsson, "The Base16, Base32, and Base64 data encodings," Internet Requests for Comments, RFC Editor, RFC 4648, October 2006, http://www.rfc-editor.org/rfc/rfc4648.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4648.txt

[27] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," Internet Requests for Comments, RFC Editor, RFC 2898, September 2000, http://www.rfc-editor.org/rfc/rfc2898.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2898.txt

[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[29] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 216–226.

[30] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020.

[31] M. D. A. Maia, V. Sobreira, K. R. Paixão, R. A. D. Amo, and I. R. Silva, "Using a sequence alignment algorithm to identify specific and common code from execution traces," in *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, 2008, pp. 6–10.

[32] H. Massalin, "Superoptimizer— A Look at the Smallest Program," *ACM SIGPLAN Notices*, vol. 22, no. 10, pp. 122–126, 10 1987.

[33] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza, "Using entropy measures for comparison of software traces," *Information Sciences*, vol. 203, pp. 59–72, oct 2012.

[34] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009.

[35] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 297–310, Mar. 2016.

[36] A. Rossberg, "WebAssembly Core Specification," W3C, Tech. Rep., Dec. 2019. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[37] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity," in *NDSS*, 2017.

[38] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A Synthesizing Superoptimizer," *arXiv preprint 1711.04422*, 2017.

[39] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[40] M. Taguinod, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward a moving target defense for web applications," in *2015 IEEE International Conference on Information Reuse and Integration*. IEEE, 2015, pp. 510–517.

[41] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2001, pp. 193–202.

[42] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proceedings of the First ACM Workshop on Moving Target Defense*, 2014, pp. 31–40.

MULTI-VARIANT EXECUTION AT THE EDGE

Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Under review

Multi-variant Execution at the Edge

Javier Cabrera Arteaga, Pierre Laperdrix, Martin Monperrus, and Benoit Baudry

Abstract—Edge-Cloud computing offloads parts of the computations that traditionally occurs in the cloud to edge nodes. The binary format WebAssembly is increasingly used to distribute and deploy services on such platforms. Edge-Cloud computing providers let their clients deploy stateless services in the form of WebAssembly binaries, which are then translated to machine code, sandboxed and executed at the edge. In this context, we propose a technique that (i) automatically diversifies WebAssembly binaries that are deployed to the edge and (ii) randomizes execution paths at runtime. Thus, an attacker cannot exploit all edge nodes with the same payload. Given a service, we automatically synthesize functionally equivalent variants for the functions providing the service. All the variants are then wrapped into a single multivariant WebAssembly binary. When the service endpoint is executed, every time a function is invoked, one of its variants is randomly selected. We implement this technique in the MEWE tool and we validate it with 7 services for which MEWE generates multivariant binaries that embed hundreds of function variants. We execute the multivariant binaries on the world-wide edge platform provided by Fastly, as part as a research collaboration. We show that multivariant binaries exhibit a real diversity of execution traces across the whole edge platform distributed around the globe.

Index Terms—Diversification, Edge-Cloud computing, Multivariant execution, WebAssembly.

1 Introduction

Edge-Cloud computing distributes a part of the data and computation to edge nodes [1], [2]. Edge nodes are servers located in many countries and regions so that Internet resources get closer to the end users, in order to reduce latency and save bandwidth. Video and music streaming services, mobile games, as well as e-commerce and news sites leverage this new type of cloud architecture to increase the quality of their services. For example, the New York Times website was able to serve more than 2 million concurrent visitors during the 2016 US presidential election with no difficulty thanks to Edge computing [3].

The state of the art of edge computing platforms like Cloudflare or Fastly use the binary format WebAssembly (aka Wasm) [4], [5] to deploy and execute on edge nodes. WebAssembly is a portable bytecode format designed to be lightweight, fast and safe [6], [7]. After compiling code to a WebAssembly binary, developers spawn an edge-enabled compute service by deploying the binary on all nodes in an Edge platform. Thanks to its simple memory and computation model, WebAssembly is considered safe [8], yet is not exempt of vulnerabilities either at the execution engine's level [9] or the binary itself [10]. Implementations in both, browsers and standalone runtimes [11], have been found to be vulnerable [10], [11], opening the door to different attacks. This means that if one node in an Edge network is vulnerable, all the others are vulnerable in the exact same manner as the same binary is replicated on each node. In other words, the same attacker payload would break all edge nodes at once. This illustrates how Edge computing is fragile with respect to systemic vulnerabilities for the whole network, like it happened on June 8, 2021 for Fastly [12].

In this work, we introduce Multivariant Execution for WebAssembly in the Edge(MEWE), a framework that generates diversified WebAssembly binaries so that no two executions in the edge network are identical. Our solution is inspired by N-variant systems [13] where diverse variants are assembled for secretless security. Here, our goal is to drastically increase the effort for exploitation through large-scale execution path randomization. MEWE operates in two distinct steps. At compile time, MEWE generates *variants* for different functions in the program. A function variant is semantically identical to the original function but structurally different, i.e., binary instructions are in different orders or have been replaced with equivalent ones. All the function variants for one service are then embedded in a single multivariant WebAssembly binary. At runtime, every time a function is invoked, one of its variant is randomly selected. This way, the actual execution path taken to provide the service is randomized each time the service is executed resulting in harder break-once-break-everywhere attacks.

We experiment MEWE with 7 services, composed of hundreds of functions. We successfully synthesize thousands of function variants, which create orders of magnitude more possible execution paths than in the original service. To determine if these new paths embedded in the service binaries are actually randomly triggered at runtime, we deploy and run them on the Fastly edge computing platform, a leading world-wide content-delivery network (CDN). We collaborated with Fastly to experiment MEWE on the actual production edge computing nodes that they provide to their clients. This means that all our experiments ran in a real-world setting, together with Fastly's customer applications, such as the New-York Times services mentioned earlier. For this experiment, we execute each multivariant binary thousands of times on every edge computing node provided by Fastly. This experiment shows that the multivariant binaries render the same service as the original, yet with highly diverse execution

J. Cabrera Arteaga is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: javierca@kth.se
P. Laperdrix. is with The French National Centre for Scientific Research (CNRS), France. E-mail: pierre.laperdrix@inria.fr
M. Monperrus is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: monperrus@kth.se
B. Baudry is with the KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: baudry.kth.se

traces.

The novelty of our contribution is as follows. First, we are the first to perform software diversification in the context of edge computing, with experiments performed on a real-world, large-scale, commercial edge computing platform (Fastly). Second, very few works have looked at software diversity for WebAssembly [14], [11], our paper contributes to proving the feasibility of this challenging endeavour.

To sum up, our contributions are:

- MEWE: a framework that builds multivariant WebAssembly binaries for edge computing, combining the automatic synthesis of semantically equivalent function variants, with execution path randomization.
- Original results on the large-scale diversification of WebAssembly binaries, at the function and execution path levels.
- Empirical evidence of the feasibility of deploying our novel multivariant execution scheme on a real-world edge-computing platform.
- A publicly available prototype system, shared for future research on the topic: https://github.com/Jacarte/MEWE.

This work is structured as follows. First, Section 2 present a background on WebAssembly and its usage in an edge-cloud computing scenario. Section 3 introduces the architecture and foundation of MEWE while Section 4 and Section 5 present the different experiments we conducted to show the feasibility of our approach. Section 6 details the Related Work, Section 7 discusses our results while Section 8 concludes this paper.

2 Background

In this section we introduce WebAssembly, as well as the deployment model that edge-cloud platforms such as Fastly provide to their clients. This forms the technical context for our work.

2.1 WebAssembly

WebAssembly is a bytecode designed to bring safe, fast, portable and compact low-level code on the Web. The language was first publicly announced in 2015 and formalized by Haas et al. [6]. Since then, most major web browsers have implemented support for the standard. Besides the Web, WebAssembly is independent of any specific hardware, which means that it can run in standalone mode. This allows for the adoption of WebAssembly outside web browsers [7], e.g., for edge computing [11].

WebAssembly binaries are usually compiled from source code like C/C++ or Rust. Listing 1 and 2 illustrate an example of a C function turned into WebAssembly. Listing 1 presents the C code of one function and Listing 2 shows the result of compiling this function into a WebAssembly module. The WebAssembly code is further interpreted or compiled ahead of time into machine code.

2.2 Web Assembly and Edge Computing

Using Wasm as an intermediate layer is better in terms of startup and memory usage, than containerization or virtualization [8], [15]. This has encouraged edge computing platforms like Cloudflare or Fastly to adopt WebAssembly

```c
int f(int x) {
    return 2 * x + x;
}
```

Listing 1: C function that calculates the quantity $2x + x$

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (func (;0;) (type 0) (param i32) (result i32)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
    (export "f" (func 0)))
```

Listing 2: WebAssembly code for Listing 1.

(Wasm) to deploy client applications in a modular and sandboxed manner [4], [5]. In addition, WebAssembly is a compact representation of code, which saves bandwidth when transporting code over the network . This allows edge-cloud platform providers to deploy the same Wasm binary, for a client application, around the world in a few seconds.

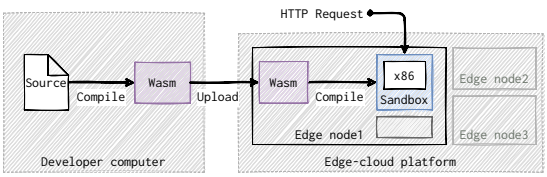


Fig. 1: Deployment to the edge. Client application developers build their application and compile it to WebAssembly before submitting to the Edge-Cloud computing platform. Then, the Wasm binary is distributed to all edge nodes, where it is compiled to a sandboxed machine code (x86 or ARM). This machine code is executed at every service request.

Client applications that are designed to be deployed on edge-cloud computing platforms are usually isolated services, having one single responsibility. This development model is known as serverless computing, or function-as-a-service [16], [11]. Figure 1 summarizes the development and deployment process for a client application to an edge-cloud platform. First, the developers of a client application implement the isolated services in a given programming language. The source code and the HTTP harness are then compiled to WebAssembly using a tool like LLVM [17] or Binaryen [18].

When client application developers deploy a WebAssembly binary for a function-as-a-service, it is sent to all edge nodes in the platform. Then, the WebAssembly binary is compiled on each node to machine code. This way, if the edge nodes have different architectures, the clients can still deploy a single WebAssembly binary, and the compilers take care of the final machine code generation step. Each binary is compiled in a way that ensures that the code runs inside an isolated sandbox.

2.3 Multivariant Execution

In 2006, security researchers of University of Virginia have laid the foundations of a novel approach to security that

consists in executing multiple variants of the same program. They called this "N-variant systems" [13]. This potent idea has been renamed soon after as "multivariant execution".

Definition 1. Multivariant execution (MVE) consists of creating multiple variants of a program and executing them in a principled way so as to improve security and reliability.

There is a wide range of realizations of MVE in different contexts. Bruschi et al. [19] and Salamat et al. [20] pioneered the idea of executing the variants in parallel. Subsequent techniques focus on MVE for mitigating memory vulnerabilities [21], [22] and other specific security problems incl. return-oriented programming attacks [23] and code injection [24]. A key design decision of MVE is whether it is achieved in kernel space [25], in user-space [26], with exploiting hardware features [27], or even throught code polymorphism [28]. Finally, one can neatly exploit the limit case of executing only two variants [29], [30]. The body of research on MVE in a distributed setting has been less researched. Notably, Voulimeneas et al. proposed a multivariant execution system by parallelizing the execution of the variants in different machines [31] for sake of efficiency.

In this paper, we propose an original kind of MVE in the context of edge computing. We generate multiple program variants, which we execute on edge computing nodes. We use the natural redundancy of Edge-Cloud computing architectures to deploy an internet-based MVE. Next section goes into the details of our procedure to generate variants and assemble them into multivariant binaries.

3 MEWE: Multivariant Execution for Edge Computing

In this section we present MEWE, a novel technique to synthesize multivariant binaries and deploy them on an edge computing platform.

3.1 Overview

The goal of MEWE is to synthesize multivariant WebAssembly binaries, according to the threat model presented in Section 3.2.1. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. The core idea of MEWE is to synthesize diversified function variants providing execution-path randomization, according to the diversity model presented in Section 3.2.3.

In Figure 2, we summarize the analysis and transformation pipeline of MEWE. We pass a bitcode to be diversified, as an input to MEWE. Analysis and transformations are performed at the level of LLVM's intermediate representation (LLVM IR), as it is the best format for us to perform our modifications (see Section 3.2.4). LLVM binaries can be obtained from any language with an LLVM frontend such as C/C++, Rust or Go, and they can easily be compiled to WebAssembly. In Step ①, the binary is passed to CROW [14], which is a superdiversifier for Wasm that generates a set of variants for the functions in the binary. Step ② packages all the variants in one single multivariant LLVM binary. In Step ③, we use a special component, called a "mixer", which augments the binary with two different components: an HTTP endpoint harness and a random generator, which are both required for executing Wasm at the edge. The harness is used to connect the program to its execution environment while the generator provides support for random execution path at runtime. The final output of Step ④ is a standalone multivariant WebAssembly binary that can be deployed on an edge-cloud computing platform. In the following sections, we describe in greater details the different stages of the workflow.

3.2 Key design choices

In this section, we introduce the main design decisions behind MEWE, starting from the threat model, to aligning the code analysis and transformation techniques.

3.2.1 Threat Model

With MEWE, we aim to defend against an attacker that wants to leak private and sensitive information by learning about the state of the system and its characteristics. These attacks include but are not limited to timing specific operations [32], [33], counting register spill/reload operations to study and attack memory side-channels [34] and performing call stack analysis. They can be performed either locally or remotely by finding a vulnerability or using shared resources in the case of a multi-tenant Edge computing server but the details of such exploitation are out of scope of this study.

3.2.2 Dangers related to Edge Computing

To benefit from the performance improvements offered by edge computing, Fastly and Cloudflare modularize their services into a set of WebAssembly functions, which are deployed on all the edge nodes provided by the edge computing platforms. Then, these services are spawned on demand. This model of distributing the exact same WebAssembly binary on hundreds of computation nodes around the world is a serious risk for the infrastructure: a malicious developer who manages to exploit one vulnerability in the binary can exploit all the compute nodes with the same attack vector [35].

3.2.3 Execution Diversification Model

MEWE is designed to randomize the execution of WebAssembly programs, via diversification transformations. Per Crane et al. those transformations are made to hinder side-channel attacks [36]. All programs are diversified with behavior preservation guarantees [14]. The core diversification strategies are:

1) *Constant Inferring*. MEWE identifies variables whose value can computed at compile time. This has an effect on program execution times [37].
2) *Call Stack Randomization*. MEWE introduces equivalent synthetic functions that are called randomly. This results in randomized call stacks, which complicates attacks based on call stack analysis [38].
3) *Inline Expansion*. MEWE inlines methods when appropriate. This also results in different call stacks, to hinder the same kind of attacks as for call stack randomization [38]. On the other hand, the number of register spill/reload operations during machine code generation changes, affecting the measurement of memory side-channels [34].

3.2.4 Diversification at the LLVM level

MEWE diversifies programs at the LLVM level. Other solutions would have been to diversify at the source code level [39], or at the native binary level, eg x86 [40]. However, the former would limit the applicability of our work. The latter is not compatible with edge computing: the top edge computing execution platforms, e.g. Cloudflare and Fastly, mostly take WebAssembly binaries as input.

LLVM, on the contrary, does not suffer from those limitations: 1) it supports different languages, with a rich ecosystem of frontends 2) it can reliably be retargeted to WebAssembly, thanks to the corresponding mature component in the LLVM toolchain. In addition, the LLVM ecosystem as a whole is very active, providing us with many different tools to facilitate our research endeavour.

3.3 Variant generation

MEWE relies on the superdiversifier CROW [14] to automatically diversify each function in the input LLVM binary (Step ①). CROW receives an LLVM module, analyzes the binary at the function block level and generates semantically equivalent variants for each function, if they exist. CROW variants are verified as semantically equivalent with an SMT solver. Here, we define a function variant as:

***Definition* 2.** Function variant: Let F be a function, F' is a function variant of F for MEWE if it is semantically equivalent (i.e., same input/output behavior), but exhibits a different internal behavior through tracing.

In Listing 3, we illustrate two semantically equivalent Wasm functions according to Definition 2. The left most listing corresponds to the Wasm module shown in Listing 2. The right most listing is a variant for this function. We can appreciate that the multiplication of the original code, in the third and four lines, is replaced by an addition, making the variant to have the same semantic but executing different instructions.

```
...
(func (;0;)
    local.get 0
    local.get 0
    i32.const 2
    i32.mul
    i32.add)
...
```

```
...
(func (;0;)
    local.get 0
    local.get 0
    local.get 0
    i32.add
    i32.add)
...
```

Listing 3: Example of two semantically equivalent functions. The left listing corresponds to the original code. The right listing shows a semantically equivalent variant.

CROW synthesizes variants by enumerative synthesis based on code transformation. The most relevant transformations are: constant inferring to replace control flow statements, arithmetic's equivalent replacement, and loop unrolling. CROW performs stacked transformations, this means that it can synthesize variants of different size, i.e., from smaller to larger variants than the original.

The variants created by CROW are artificially synthesized from the original binary. CROW checks for semantic equivalence of both codes, original and variant using the symbolic execution. If the behavior of the variant is not the reference behavior, the variant is discarded. This means that, after Step ①, the variant is necessarily equivalent to the original program.

3.4 Combining variants into multivariant functions

Step ② of MEWE consists in combining the variants generated for the original functions, into into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function for which we generate variants. A dispatcher function is a synthetic function that is in charge of choosing a variant at random, every time the original function is invoked during the execution. The random invocation of different variants at runtime is a known randomization technique, for example used by Lettner et al. with sanitizers [41].

With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 3. Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E \rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [42], where the nodes are typed. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.

In Figure 3, we show the original static call graph for program bin2base64 (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original bin2base64 includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

In Listing 4, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 3. It first calls the random generator, which returns a value that is then used to invoke a specific function variant. It should be noted that the dispatcher function is constructed using the same signature as the original function.

We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [11]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [43]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

3.5 MEWE's Mixer

The MEWE mixer has four specific objectives: wrap functions as HTTP endpoints, link the LLVM multivariant binary, inject a random generator and merge all these components into a multivariant WebAssembly binary.

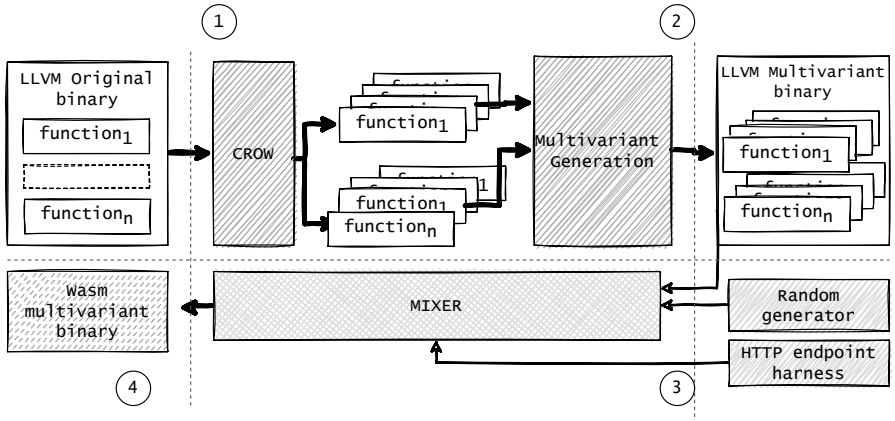


Fig. 2: Overview of MEWE. It takes as input the LLVM binary representation of a service composed of multiple functions. It first generates a set of functionally equivalent variants for each function in the binary and then generates a LLVM multivariant binary composed of all the function variants as well as dispatcher functions in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and an edge specific HTTP harness, in order to produce a WebAssembly multivariant binary accessible through an HTTP endpoint and ready to be deployed to the edge.

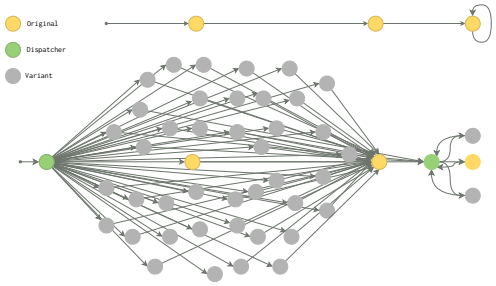


Fig. 3: Example of two static call graphs for the bin2base64 endpoint of libsodium. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

```
define internal i32 @b64_byte2urlsafe_char(i32 %0) {
  entry:
    %1 = call i32 @discriminate(i32 3)
    switch i32 %1, label %end [
      i32 0, label %case_43_
      i32 1, label %case_44_
    ]
  case_43_: ; preds = %entry
    %2 = call i32 @b64_byte_to_urlsafe_char_43_(%0)
    ret i32 %2
  case_44_: ; preds = %entry
    %3 = call i32 @b64_byte_to_urlsafe_char_44_(%0)
    ret i32 %3
  end: ; preds = %entry
    %4 = call i32 @b64_byte2urlsafe_char_original(%0)
    ret i32 %4
}
```

Listing 4: Dispatcher function embedded in the multivariant binary of the bin2base64 endpoint of libsodium, which corresponds to the rightmost green node in Figure 3.

We use the Rustc compiler[1] to orchestrate the mixing because Rustc is a compiler able to merge custom Rust source code with arbitratry-compatible LLVM binary producing a final Wasm binary. For the generator, we rely on WASI's specification [44] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed. For the HTTP harnesses, since our edge computing use case is based on the Fastly infrastructure, we rely on the Fastly API[2] to transform our Wasm binaries into HTTP endpoints. The harness enables a function to be called as an HTTP request and to return a HTTP response. Throughout this paper, we refer to an endpoint as the closure of invoked functions when the entry point of the WebAssembly binary is executed.

3.6 Multivariant Binary Execution at the Edge

When a WebAssembly binary is deployed on an edge platform, it is translated to machine code on the fly. For our experiment, we deploy on the production edge nodes of Fastly. This edge computing platform uses Lucet, a native WebAssembly compiler and runtime, to compile and run the deployed Wasm binary [3]. Lucet generates x86 machine code and ensures that the generated machine code executes inside a secure sandbox, controlling memory isolation.

Figure 4 illustrates the runtime behavior of the original and the multivariant binary, when deployed on an

1. https://doc.rust-lang.org/rustc/what-is-rustc.html
2. https://docs.rs/crate/fastly/0.7.3
3. https://github.com/bytecodealliance/lucet

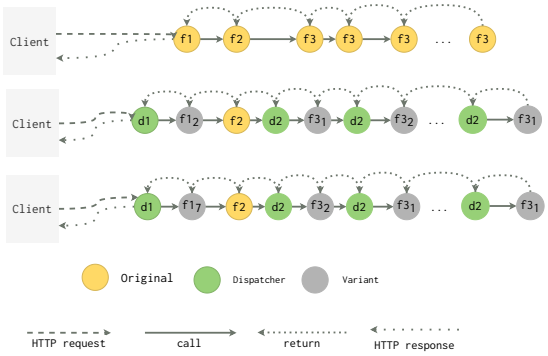


Fig. 4: Top: an execution trace for the `bin2base64` endpoint. Middle and bottom: two different execution traces for the multivariant `bin2base64`, exhibited by two different requests with exactly the same input.

Edge node. The top most diagram illustrates the execution trace for the original of the endpoint `bin2base64`. When the HTTP request with the input `"HelloWorld!"` is received, it invokes functions $f1$, $f2$ followed by 27 recursive calls of function $f3$. Then, the endpoint sends the result `"0x000xccv0x10x00b3Jsx130x000x00 0x00xpopAHRvdGE="` of its base64 encoding in an HTTP response.

The two diagrams at the bottom of Figure 4 illustrate two executions traces observed through two different requests to the endpoint `bin2base64`. In the first case, the request first triggers the invocation of dispatcher $d1$, which randomly decides to invoke the variant $f1_2$; then $f2$, which has not been diversified by MEWE, is invoked; then the recursive invocations to $f3$ are replaced by iterations over the execution of dispatcher $d2$ followed by a random choice of variants of $f3$. Eventually the result is computed and sent back as an HTTP response. The second execution trace of the multivariant binary shows the same sequence of dispatcher and function calls as the previous trace, and also shows that for a different requests, the variants of $f1$ and $f3$ are different.

The key insights from these figures are as follows. First, from a client's point of view, a request to the original or to a multivariant endpoint, is completely transparent. Clients send the same data, receive the same result, through the same protocol, in both cases. Second, this figure shows that, at runtime, the execution paths for the same endpoint are different from one execution to another, and that this randomization process results from multiple random choices among function variants, made through the execution of the endpoint.

3.7 Implementation

The multivariant combination (Step ②) is implemented in 942 lines of C++ code. Its uses the LLVM 12.0.0 libraries to extend the LLVM standard linker tool capability with the multivariant generation. MEWE's Mixer (Step ③) is implemented as an orchestration of the rustc and the WebAssembly backend provided by CROW. For sake of open science and for fostering research on this important topic, the code of MEWE is made publicly available on GitHub: https://github.com/Jacarte/MEWE.

4 Experimental Methodology

In this section we introduce our methodology to evaluate MEWE. First, we present our research questions and the services with which we experiment the generation and the execution of multivariant binaries. Then, we detail the methodology for each research question.

4.1 Research questions

To evaluate the capabilities of MEWE, we formulate the following research questions:

RQ1: **(Multivariant Generation) How much diversity can MEWE synthesize and embed in a multivariant binary ?** MEWE packages function variants in multivariant binaries. With this first question, we aim at measuring the amount of diversity that MEWE can synthesize in the call graph of a program.

RQ2: **(Intra MVE) To what extent does MEWE achieve multivariant executions on an edge compute node?** With this question we assess the ability of MEWE to produce binaries that actually exhibit random execution paths when executed on one edge node.

RQ3: **(Internet MVE) To what extent does MEWE achieve multivariant execution over the worldwide Fastly infrastructure?** We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes in order to assess MVE at a worldwide scale.

RQ4: **What is the impact of the proposed multi-version execution on timing side-channels?** MEWE generates binaries that embed a multivariant behavior. We measure to what extent MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

The core of the validation methodology for our tool MEWE, consists in building multivariant binaries for several, relevant endpoints and to deploy and execute them on the Fastly edge-cloud platform.

4.2 Study subjects

We select two mature and typical edge-cloud computing projects to study the feasibility of MEWE. The projects are selected based on: suitability for diversity synthesis with CROW (the projects should have the ability to collect their modules in LLVM intermediate representation), suitability for deployment on the Fastly infrastructure (the project should be easily portable Wasm/WASI and compatible with the Rust Fastly API), low chances to hit execution paths with no dispatchers and possibility to collect their execution runtime information (the endpoints should execute in a reasonable time of maximum 1 second even with the overhead of instrumentation). The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust.

Name	#Endpoints	#Functions	#Instr.
libsodium https://github.com/jedisct1/libsodium	5	62	6187
qrcode-rust https://github.com/kennytm/qrcode-rust	2	1840	127700

TABLE 1: Selected projects to evaluate MEWE: project name; the number of endpoints in the project that we consider for our experiments, the total number of functions to implement the endpoints, and the total number of WebAssembly instructions in the original binaries.

In Table 1, we summarize some key metrics that capture the relevance of the selected projects. The table shows the project name with its repository address, the number of selected endpoints for which we build multivariant binaries, the total number of functions included in the endpoints and the total number of Wasm instructions in the original binary. Notice that, the metadata is extracted from the Wasm binaries before they are sent to the edge-cloud computing platform, thus, the number of functions might be not the same in the static analysis of the project source code

4.3 Experimentation platform

We run all our experiments on the Fastly edge computing platform. We deploy and execute the original and the multivariant endpoints on 64 edge nodes located around the world[4]. These edge nodes usually have an arbitrary and heterogeneous composition in terms of architecture and CPU model. The deployment procedure is the same as the one described in Section 2.2. The developers implement and compile their services to WebAssembly. In the case of Fastly, the WebAssembly binaries need to be implemented with the Fastly platform API specification so they can properly deal with HTTP requests. When the compiled binary is transmitted to Fastly, it is translated to x86 machine code with Lucet, which ensures the isolation of the service.

4.4 RQ1 Multivariant diversity

We run MEWE on each endpoint function of our 7 endpoints. In this experiment, we bound the search for function variant with timeout of 5 minutes per function. This produces one multivariant binary for each endpoint. To answer RQ1, we measure the number of function variants embedded in each multivariant binary, as well as the number of execution paths that are added in the mutivariant call graphs, thanks to the function variants.

4.5 RQ2 Intra MTD

We deploy the multivariant binaries of each of the 7 endpoints presented in Table 2, on the 64 edge nodes of Fastly. We execute each endpoint, multiple times on each node, to measure the diversity of execution traces that are exhibited by the multivariant binaries. We have a time budget of 48 hours for this experiment. Within this timeframe, we can query each endpoint 100 times on each node. Each query on the same endpoint is performed with the same input value. This is to guarantee that, if we observe different traces for different executions, it is due to the presence of multiple function variants. The input values are available as part of our reproduction package.

For each query, we collect the execution trace , i.e., the sequence of function names that have been executed when triggering the query. To observe these traces, we instrument the multivariant binaries to record each function entrance.

To answer RQ2, we measure the number of unique execution traces exhibited by each multivariant binary, on each separate edge node. To compare the traces, we hash them with the `sha256` function. We then calculate the number of unique hashes among the 100 traces collected for an endpoint on one edge node. We formulate the following definitions to construct the metric for RQ3.

Metric 1. Unique traces: $R(n, e)$. Let $S(n, e) = \{T_1, T_2, ..., T_{100}\}$ be the collection of 100 traces collected for one endpoint e on an edge node n, $H(n, e)$ the collection of hashes of each trace and $U(n, e)$ the set of unique trace hashes in $H(n, e)$. The uniqueness ratio of traces collected for edge node n and endpoint e is defined as

$$R(n, e) = \frac{|U(n, e)|}{|H(n, e)|}$$

The inputs that we pass to execute the endpoints at the edge and the received output for all executions are available in the reproduction repository at https://github.com/Jacarte/MEWE.

4.6 RQ3 Inter MTD

We answer RQ3 by calculating the normalized Shannon entropy for all collected execution traces for each endpoint. We define the following metric.

Metric 2. Normalized Shannon entropy: $E(e)$ Let e be an endpoint, $C(e) = \cdot_{n=0}^{64} H(n, e)$ be the union of all trace hashes for all edge nodes. The normalized Shannon Entropy for the endpoint e over the collected traces is defined as:

$$E(e) = -\Sigma \frac{p_x * log(p_x)}{log(|C(e)|)}$$

Where p_x is the discrete probability of the occurrence of the hash x over $C(e)$.

Notice that we normalize the standard definition of the Shannon Entropy by using the perfect case where all trace hashes are different. This normalization allows us to compare the calculated entropy between endpoints. The value of the metric can go from 0 to 1. The worst entropy, value 0, means that the endpoint always perform the same path independently of the edge node and the number of times the trace is collected for the same node. On the contrary, 1 for the best entropy, when each edge node executes a different path every time the endpoint is requested.

The Shannon Entropy gives the uncertainty in the outcome of a sampling process. If a specific trace has a high frequency

4. The number of nodes provided in the whole platform is 72, we decided to keep only the 64 nodes that remained stable during our experimentation.

of appearing in part of the sampling, then it is certain that this trace will appear in the other part of the sampling.

We calculate the metric for the 7 endpoints, for 100 traces collected from 64 edge nodes, for a total of 6400 collected traces per endpoint. Each trace is collected in a round robin strategy, i.e., the traces are collected from the 64 edge nodes sequentially. For example, we collect the first trace from all nodes before continuing to the collection of the second trace. This process is followed until 100 traces are collected from all edge nodes.

4.7 RQ4 Timing side-channels

For each endpoint listed in Table 2, we measure the impact of MEWE on timing. For this, we use the following metric:

Metric 3. Execution time: For a deployed binary on the edge, the execution time is the time spent on the edge to execute the binary.

Note that edge-computing platforms are, by definition, reached from the Internet. Consequently, there may be latency in the timing measurement due to round-trip HTTP requests. This can bias the distribution of measured execution times for the multivariant binary. To avoid this bias, we instrument the code to only measure the execution on the edge nodes.

We collect 100k execution times for each binary, both the original and multivariant binaries. We perform a Mann-Withney U test [45] to compare both execution time distributions. If the P-value is lower than 0.05, two compared distributions are different.

5 Experimental Results

5.1 RQ1 Results: Multivariant generation

We use MEWE to generate a multivariant binary for each of the 7 endpoints included in our 2 study subjects. We then calculate the number of diversified functions, in each endpoint, as well as how they combine to increase the number of possible execution paths in the static call graph for the original and the multivariant binaries.

The sections 'Original binary' and 'Multivariant WebAssembly binary' of Table 2 summarize the key data for RQ1. In the 'Original binary' section, the first column (#Functions) gives the number of functions in the original binary and the second column (#Paths) gives the number of possible execution paths in the original static call graph. The 'Multivariant WebAssembly binary' section first shows the number of each type of nodes in the multivariant call graph: #Non div. is the number of original functions that could not be diversified by MEWE, #Dispatchers is the number of dispatcher nodes generated by MEWE for each function that was successfully diversified, and #Variants is the total number of function variants generated by MEWE. The last column of this section is the number of possible execution paths in the static multivariant call graph.

For all 7 endpoints, MEWE was able to diversify several functions and to combine them in order to increase the number of possible execution paths in several orders of magnitude. For example, in the case of the `encrypt` function of libsodium, the original binary contains 23 functions that can be combined in 4 different paths. MEWE generated a total of 56 variants for 5 of the 23 functions. These variants, combined with the 18 original functions in the multivariant call graph, form 325 execution paths. In other words, the number of possible ways to achieve the same encryption function has increased from 4 to 325, including dispatcher nodes that are in charge of randomizing the choice of variants at 5 different locations of the call graph. This increased number of possible paths, combined with random choices, made at runtime, increases the effort a potential attacker needs to guess what variant is executed and hence what vulnerability she can exploit.

We have observed that there is no linear correlation between the number of diversified functions, the number of generated variants and the number of execution paths. We have manually analyzed the endpoint with the largest number of possible execution paths in the multivariant Wasm binary: `qr_str` of qrcode-rust. MEWE generated 2092 function variants for this endpoint. Moreover, MEWE inserted 17 dispatchers in the call graph of the endpoint. For each dispatcher, MEWE includes between 428 and 3 variants. If the original execution path contains function for which MEWE is able to generate variants, then, there is a combinatorial explosion in the number of execution paths for the generated Wasm multivariant module. The increase of the possible execution paths theoretically augments the uncertainty on which one to perform, in the latter case, approx. 140 000 times. As Cabrera and colleagues observed [14] for CROW, a large presence of loops and arithmetic operations in the original function code leverages to more diversification.

Looking at the #Dispatchers and #Variants columns of the 'Multivariant WebAssembly binary' section of Table 2, we notice that the number of variants generated per function greatly varies. For example, for both the `invert` and the `bin2base64` functions of Libsodium, MEWE manages to diversify 2 functions (reflected by the presence of 2 dispatcher nodes in the multivariant call graph). Yet, MEWE generates a total of 125 variants for the 2 functions in `invert`, and only 47 variants for the 2 functions in `bin2base64`. The main reason for this is related to the complexity of the diversified functions, which impacts the opportunities for the synthesis of code variations.

Columns #Originals of the 'Multivariant WebAssembly binary' section of Table 2 indicates that, in each endpoint, there exists a number of functions for which MEWE did not manage to generate variants. We identify three reasons for this, related to the diversification procedure of CROW, used by MEWE to diversify individual functions. First, some functions cannot be diversified by CROW, e.g., functions that wrap only memory operations, which are oblivious to CROW diversification technique. Second, the complexity of the function directly affects the number of variants that CROW can generate. Third, the diversification procedure of CROW is essentially a search procedure, which results are directly impacted by the tie budget for the search. In all experiments we give CROW 5 minutes maximum to synthesize function variants, which is a low budget for many functions. It is important to notice that, the successful diversification of some functions in each endpoint, and their combination within the call graph of the endpoint, dramatically increases the number of possible paths that can triggered for multivariant executions.

	Original binary		Multivariant WebAssembly binary			
Endpoint	#Functions	#Paths	#Non diversified	#Dispatchers	#Variants	#Paths
libsodium						
encrypt	23	4	18	5	56	325
decrypt	20	3	16	5	49	84
random	8	2	6	2	238	12864
invert	8	2	6	2	125	2784
bin2base64	3	2	1	2	47	172
qrcode-rust						
qr_str	982	$688{*}10^6$	965	17	2092	$97{*}10^{12}$
qr_image	858	$1.4{*}10^6$	843	15	2063	$3{*}10^9$

TABLE 2: Static diversity generated by MEWE, measured on the static call graphs of the WebAssembly binaries, and the preservation of this diversity after translation to machine code. The table is structured as follows: Endpoint name; number of functions and numbers of possible paths in the original WebAssembly binary call graph; number of non diversified functions, number of created dispatchers (one per diversified functions), total number of function variants and number of execution paths in the multivariant WebAssembly binary call graph.

Answer to RQ1

MEWE dramatically increases the number of possible execution paths in the multivariant WebAssembly binary of each endpoint. The large number of possible execution paths, combined with multiple points of random choice in the multivariant call graph thwart the prediction of which path will be taken at runtime.

5.2 RQ2 Results: Intra MTD

To answer RQ2, we execute the multivariant binaries of each endpoint, on the Fastly edge-cloud infrastructure. We execute each endpoint 100 times on each of the 64 Fastly edge nodes. All the executions of a given endpoint are performed with the same input. This allows us to determine if the execution traces are different due to the injected dispatchers and their random behavior. After each execution of an endpoint, we collect the sequence of invoked functions, i.e., the execution trace. Our intuition is that the random dispatchers combined with the function variants embedded in a multivariant binary are very likely to trigger different traces for the same execution, i.e., when an endpoint is executed several times in a row with the same input and on the same edge node. The way both the function variants and the dispatchers contribute to exhibiting different execution traces is illustrated in Figure 4.

Figure 5 shows the ratio of unique traces exhibited by each endpoint, on each of the 64 separate edge nodes. The X corresponds to the edge nodes. The Y axis gives the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point in the Z axis representing Metric 1 over 100 execution traces.

For all edge nodes, the ratio of unique traces is above 0.38. In 6 out of 7 cases, we have observed that the ratio is remarkably high, above 0.9. These results show that MEWE generates multivariant binaries that can randomize execution paths at runtime, in the context of an edge node. The randomization dispatchers, associated to a significant number of function variants greatly reduce the certainty about which computation is performed when running a specific input with a given input value.

Let's illustrate the phenomenon with the endpoint `invert`. The endpoint `invert` receives a vector of integers and returns its inversion. Passing a vector of integers with 100 elements as input, $I = [100, ..., 0]$, results in output $O = [0, ..., 100]$. When the endpoint executes 100 times with the same input on the original binary, we observe 100 times the same execution trace. When the endpoint is executed 100 times with the same input I on the multivariant binary, we observe between 95 and 100 unique execution traces, depending on the edge node. Analyzing the traces we observe that they include only two invocations to a dispatcher, one at the start of the trace and one at the end. The remaining events in the trace are fixed each time the endpoint is executed with the same input I. Thus, the maximum number of possible unique traces is the multiplication of the number of variants for each dispatcher, in this case $29 \times 96 = 2784$. The probability of observing the same trace is $1/2784$.

For multivariant binaries that embed only a few variants, like in the case of the `bin2base64` endpoint, the ratio of unique traces per node is lower than for the other endpoints. With the input we pass to `bin2base64`, the execution trace includes 57 function calls. We have observed that, only one of these calls invokes a dispatcher, which can select among 41 variants. Thus, probability of having the same execution trace twice is $1/41$.

Meanwhile, `qr_str` embeds thousands of variants, and the input we pass triggers the invocation of 3M functions, for which 210666 random choices are taken relying on 17 dispatchers. Consequently, the probability of observing the same trace twice is infinitesimal. Indeed, all the executions of `qr_str` are unique, on each separate edge node. This is shown in Figure 5, where the ratio of unique traces is 1 on all edge nodes.

Answer to RQ2

Repeated executions of a multivariant binary with the same input on an individual edge node exhibits diverse execution traces. MEWE successfully synthesizes multivariant binaries that trigger diverse execution paths at runtime, on individual edge nodes.

5.3 RQ3 Results: Internet MTD

To answer RQ3, we build the union of all the execution traces collected on all edge nodes for a given endpoint. Then, we compute the normalized Shannon Entropy over this set for each endpoint (Metric 2). Our goal is to determine whether the diversity of execution traces we observed on individual

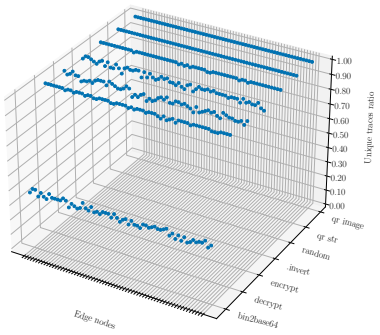


Fig. 5: Ratio of unique execution traces for each endpoint on each edge node. The X axis illustrates the edge nodes. The Y axis annotates the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point representing the Metric 1 value in a set of 100 collected execution traces.

Endpoint	Entropy	MTL	σ	#ED	#RCh
libsodium					
encrypt	0.87	816	0	5	4M
decrypt	0.96	440	0	5	2M
random	0.98	15	5	2	12800
invert	0.87	7343	0	2	12800
bin2base64	0.42	57	0	1	6400
qrcode-rust					
qr_str	1.00	3045193	0	17	1348M
qr_image	1.00	3015450	0	15	1345M

TABLE 3: Execution trace diversity over the Fastly edge-cloud computing platform. The table is formed of 6 columns: the name of the endpoint, the normalized Shannon Entropy value (Metric 2), the median size of the execution traces (MTL), the standard deviation for the trace lengths the number of executed dispatchers (#ED) and the number of total random choices taken during all the 6400 executions (#RCh).

nodes in RQ3, actually generalizes to the whole edge-cloud infrastructure. Depending on many factors, such as the random number generator or a bug in the dispatcher, it could happen that we observe different traces on individual nodes, but that the set of traces is the same on all nodes. With RQ4 we assess the ability of MEWE to exhibit multivariant execution at a global scale.

Table 3 provides the data to answer RQ3. The second column gives the normalized Shannon Entropy value (Metric 2). Columns 3 and 4 give the median and the standard deviation for the length of the execution traces. Columns 5 and 6 give the number of dispatchers that are invoked during the execution of the endpoint (#ED) and the total number of invocations of these endpoints (#Rch). These last two columns indicate to what extent the execution paths are actually randomized at runtime. In the cases of `invert` and `random`, both have the same number of taken random choices. However, the number of variants to chose in `random` are larger, thus, the entropy, is larger than `invert`.

Overall, the normalized Shannon Entropy is above 42%. This is evidence that the multivariant binaries generated by MEWE can indeed exhibit a high degree of execution trace diversity, while keeping the same functionality. The number of randomization points along the execution paths (#Rch) is at the core of these high entropy values. For example, every execution of the `encrypt` endpoint triggers 4M random choices among the different function variants embedded in the multivariant binaries. Such a high degree of randomization is essential to generate very diverse execution traces.

The `bin2base64` endpoint has the lowest level of diversity. As discussed in RQ2, this endpoint is the one that has the least variants and its execution path can be randomized only at one point. The low level of unique traces observed on individual nodes is reflected at the system wide scale with a globally low entropy.

For both `qr_str` and `qr_image` the entropy value is 1.0. This means that all the traces that we observe for all the executions of these endpoints are different from each other. In other words, someone who runs these services over and over with the same input cannot know exactly what code will be executed in the next execution. These very high entropy values are made possible by the millions of random choices that are made along the execution paths of these endpoints.

While there is a high degree of diversity among the traces exhibited by each endpoint, they all have the same length, except in the case of `random`. This means that the entropy is a direct consequence of the invocations of the dispatchers. In the case of `random`, it naturally has a non-deterministic behavior. Meanwhile, we observe several calls to dispatchers in during the execution of the multivariant binary, which indicates that MEWE can amplify the natural diversity of traces exhibited by `random`. For each endpoint, we managed to trigger all dispatchers during its execution. There is a correlation between the entropy and the number of random choices (Column #RChs) taken during the execution of the endpoints. For a high number of dispatchers, and therefore random choices, the entropy is large, like the cases of `qr_str` and `qr_image` show. The contrary happens to `bin2base64` where its multivariant binary contains only one dispatcher.

Answer to RQ3

At the internet scale of the Edge platform, the multivariant binaries synthesized by MEWE exhibit a massive diversity of execution traces, while still providing the original service. It is virtually impossible for an attacker to predict which is taken for a given query.

5.4 RQ4 Results: Timing side-channels

For each endpoint used in RQ1, we compare the execution time distributions for the original binary and the multivariant binary. All distributions are measured on 100k executions. In Table 4, we show the execution time for the original endpoints and their corresponding multivariant. The table is structured in two sections. The first section shows the endpoint name, the

	Original bin.		Multivariant Wasm	
Endpoint	Median (μs)	σ	Median (μs)	σ
libsodium				
encrypt	7	5	217	43
decrypt	13	6	225	47
random	16	7	232	53
invert	119	34	341	65
bin2base64	10	5	215	35
qrcode-rust				
qr_str	3,117	418	492,606	36,864
qr_image	3,091	412	512,669	41,718

TABLE 4: Execution time distributions for 100k executions, for the original endpoints and their corresponding multivariants. The table is structured in two sections. The first section shows the endpoint name, the median execution time and its standard deviation for the original endpoint. The second section shows the median execution time and its standard deviation for the multivariant WebAssembly binary.

median and standard deviation of the original endpoint. The second section shows the median and the standard deviation for the execution time of the corresponding multivariant binary.

We also observe that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 6, each subplot represents the distribution for a single endpoint, with the colors blue and green representing the original and multivariant binary respectively. These plots reveal that the execution times are indeed spread over a larger range of values compared to the original binary. This is evidence that execution time is less predictable for multivariant binaries than for the original ones.

We evaluate to what extent a specific variant can be detected by observing the execution time distribution. This evaluation is based on the measurement with one endpoint. For this, we choose endpoint `bin2base64` because it is the end point that has the least variants and the least dispatchers, which is the most conservative assumption.

We dissect the collected execution times for the `bin2base64` endpoint, grouping them by execution path. In Figure 7, each opaque curve represents a cumulative execution time distribution of a unique execution path out of the 41 observed. We observe that no specific distribution is remarkably different from another one. Thus, no specific variant can be inferred out of the projection of all execution times like the ones presented in Figure 6. Nevertheless, we calculate a Mann-Whitney test for each pair of distributions, 41×41 pairs. For all cases, there is no statistical evidence that the distributions are different, $P > 0.05$.

Recall that the choice of function variant is randomized at each function invocation, and the variants have different execution times as a consequence of the code transformations, i.e., some variants execute more instructions than others. Consequently, attacks relying on measuring precise execution times of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

We note that the execution times are slower for multivariant binaries. Being under 500 ms in general, this does not represent a threat to the applicability of multivariant execution at the edge. Yet, it calls for future optimization research.

Answer to RQ4

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. MEWE contributes to mitigate potential attacks based on predictable execution times.

6 Related Work

Our work is in the area of software diversification for security, a research field discovered by researchers Forrest [46] and Cohen [47]. We contribute a novel technique for multivariant execution, and discuss related work in Section 2. Here, we position our contribution with respect to previous work on randomization and security for WebAssembly.

6.1 Related Work on Randomization

A randomization technique creates a set of unique executions for the very same program [48]. Seminal works include instruction-set randomization [49], [50] to create a unique mapping between artificial CPU instructions and real ones. This makes it very hard for an attacker ignoring the key to inject executable code. Compiler-based techniques can randomly introduce NOP and padding to statically diversify programs. [51] have explored how to use NOP and it breaks the predictability of program execution, even mitigating certain exploits to an extent.

Chew and Song [52] target operating system randomization. They randomize the interface between the operating system and the user applications: the system call numbers, the library entry points (memory addresses) and the stack placement. All those techniques are dynamic, done at runtime using load-time preprocessing and rewriting. Bathkar et al. [48], [53] have proposed three kinds of randomization transformations: randomizing the base addresses of applications and libraries memory regions, random permutation of the order of variables and routines, and the random introduction of random gaps between objects. Dynamic randomization can address different kinds of problems. In particular, it mitigates a large range of memory error exploits. Recent work in this field include stack layout randomization against data-oriented programming [54] and memory safety violations [55], as well as a technique to reduce the exposure time of persistent memory objects to increase the frequency of address randomization [56].

We contribute to the field of randomization, at two stages. First, we automatically generate variants of a given program, which have different WebAssembly code and still behave the same. Second, we randomly select which variant is executed at runtime, creating a multivariant execution scheme that

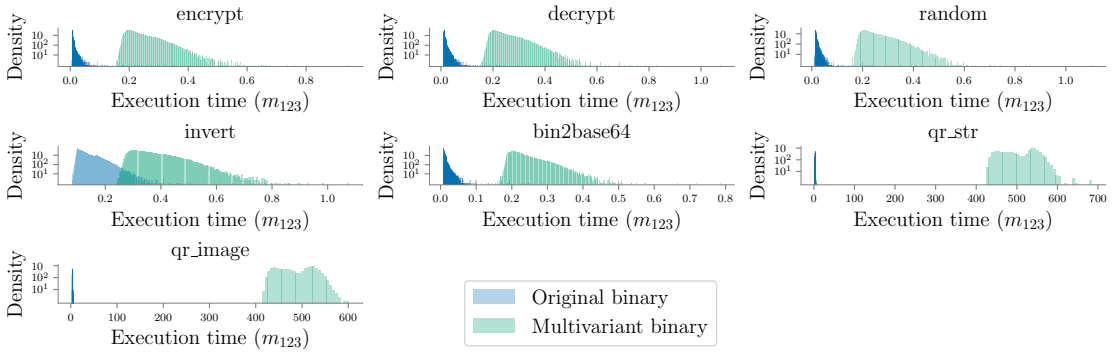


Fig. 6: Execution time distributions. Each subplot represents the distribution for a single endpoint, blue for the original endpoint and green for the multivariant binary. The X axis shows the execution time in milliseconds and the Y axis shows the density distribution in logarithmic scale.

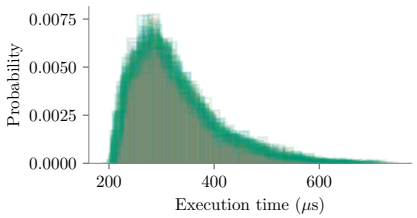


Fig. 7: Execution time distributions for the `bin2base64` endpoint. Each opaque curve represents an execution time distribution of a unique execution path out of the 41 observed.

randomizes the observable execution trace at each run of the program.

Davi et al. proposed Isomeron [57], an approach for execution-path randomization. Isomeron simultaneously loads the original program and a variant. While the program is running, Isomeron continuously flips a coin to decide which copy of the program should be executed next at the level of function calls. With this strategy, a potential attacker cannot predict whether the original or the variant of a program will execute. MEWE proposes two key novel contributions. First, our code diversification step can generate variants of complex control flow structures by inferring constants or loop unrolling. Second, MEWE interconnects hundreds of variants and several randomization dispatchers in a single binary, increasing by orders of magnitude the runtime uncertainty about what code will actually run, compared to the choice among 2 variants proposed by Isomeron.

6.2 Related work on WebAssembly Security

The reference piece about WebAssembly security is by Lehmann et al. [10], which presents three attack primitives. Lehmann et al. have then followed up with a large-scale empirical study of WebAssembly binaries [58]. Narayan et al. [11] remark that the security model of WebAssembly is vulnerable to Spectre attacks. This means that WebAssembly sandboxes may be hijacked and leak memory. They propose to modify the Lucet compiler used by Fastly to incorporate LLVM fence instructions[5] in the machine code generation, trying to avoid speculative execution mistakes. Johnson et al. [43], on the other hand, propose fault isolation for WebAssembly binaries, a technique that can be applied before being deployed to the edge-cloud platforms. Stievenart et al. [59] design a static analysis dedicated to information flow problems. Bian et al. [60] performs runtime monitoring of WebAssembly to detect cryptojacking. The main difference with our work is that our defense mechanism is larger in scope, meant to tackle "yet unknown" vulnerabilities. Notably, MEWE is agnostic from the last-step compilation pass that translates Wasm to machine code, which means that the multivariant binaries can be deployed on any edge-cloud platform that can receive WebAssembly endpoints, regardless of the underlying hardware.

7 Discussion

Specialising, optimizing, improving performance In Section 4 we validated the key features of MEWE: the automatic generation of multivariant binaries, which exhibit random execution paths at runtime. Several aspects of these procedures can be optimized. For example, the generated code can be optimized by inline function variants in the dispatchers. This minimal change will decrease the number of function calls. On the other hand, the number of times a dispatcher is called can be bound. As discussed in RQ3 and RQ4, the dispatchers are massively invoked at runtime, which is great for randomization, but also a challenge with respect to the execution time of the services.

Fuzzing and security The diversification created by MEWE can unleash hidden behaviors in compilers like Lucet. One of the biggest challenges in fuzzing compilers is the ability to reach latter stages in the machine code generation pipeline. By generating several functionally equivalent, and yet different variants, deeper bugs can be discovered. During

5. https://llvm.org/doxygen/classllvm_1_1FenceInst.html

SCALABLE COMPARISON OF JAVASCRIPT V8 BYTECODE TRACES

Javier Cabrera-Arteaga, Martin Monperrus, Benoit Baudry
SPLASH 2019, VMIL

Scalable Comparison of JavaScript V8 Bytecode Traces

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Abstract

The comparison and alignment of runtime traces are essential, e.g., for semantic analysis or debugging. However, naive sequence alignment algorithms cannot address the needs of the modern web: (i) the bytecode generation process of V8 is not deterministic; (ii) bytecode traces are large.

We present STRAC, a scalable and extensible tool tailored to compare bytecode traces generated by the V8 JavaScript engine. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk. STRAC can identify semantically equivalent web pages and is capable of processing huge V8 bytecode traces whose order of magnitude matches today's web like https://2019.splashcon.org, which generates approx. 150k of V8 bytecode instructions.

CCS Concepts • **Information systems → World Wide Web**; • **Theory of computation → Program semantics**; • **Software and its engineering →** *Interpreters*; *Source code generation*; *Designing software*.

Keywords V8, Sequence alignment, JavaScript, Bytecode, Similarity measurement

ACM Reference Format:
Javier Cabrera Arteaga, Martin Monperrus, and Benoit Baudry. 2019. Scalable Comparison of JavaScript V8 Bytecode Traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361228

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361228

1 Introduction

Runtime traces record the execution of programs. This information captures the dynamics of programs and can be used to determine semantic similarity [29], to detect abnormal program behavior [8], to check refactoring correctness [22] or to infer execution models [1]. In many cases, this is achieved by comparing execution traces, e.g. comparing the traces of the original program and the refactored one. The comparison of program traces can be based on information retrieval [17], tree differencing [9, 27] and sequence alignment [2, 11]. In this paper, we focus on the latter, in order to compare sequences of V8 bytecode instructions resulting from the execution of JavaScript code.

V8 is an open source, high-performance JavaScript engine. For debugging purposes, it provides powerful facilities to export page execution information [21], including intermediate internal bytecode called the V8 bytecode [4].

Due to the dynamic nature of the Web, we observe that the bytecode generation process of V8 is not deterministic. For example, visiting the same page several times results in different V8 bytecode traces every time. This nondeterminism is a key challenge for sequence alignment approaches, even if they perform well on deterministic program traces [10]. Besides, V8 bytecode traces are large. Naive sequence alignment algorithms are time and space quadratic on trace sizes and do not scale to V8 bytecode traces. To illustrate this scaling problem, let us consider a simple query to https://2019.splashcon.org: it generates between 139555 and 162558 V8 bytecode instructions, and aligning two traces of such size, requires approximately 150GB of memory[1]. This memory requirement is not realistic for trace analysis tasks on developer's personal computers or servers. The key challenge that we address in this work is to provide a trace comparison tool that scales to V8 bytecode traces.

In this paper, we present STRAC (Scalable Trace Comparison), a scalable and extensible tool tailored to compare bytecode traces from the V8 JavaScript engine. STRAC implements an optimized version of the DTW algorithm [18]. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk.

Our experiments compare STRAC with 6 other publicly-available implementations of DTW. The comparison involves

[1] In this paper, memory means RAM.

100 pairs of V8 bytecode traces collected over 6 websites. Our experimental results show that 1) STRAC can identify semantically equivalent web pages and 2) STRAC is capable of processing big V8 bytecode traces whose order of magnitude matches today's web.

To sum up, our contributions are:

- An analysis of the challenges for analyzing browser traces, due to the JavaScript engine internals and the randomness of the environment. We explain and show examples of how the same browser query can generate two different V8 bytecode traces.
- A tool called STRAC that implements the popular alignment algorithm DTW in a scalable way, publicly available at https://github.com/KTH/STRAC .
- A set of experiments comparing 100 V8 bytecode traces collected over 6 real world websites:google.com, kth.se, github.com, wikipedia.org, 2019.splashcon.org and youtube.com. Our experiments show that STRAC copes with the non-deterministic traces and is significantly faster than state-of-the-art tools.

The paper is structured as follows. First we introduce a background of V8 bytecode generation non-determinism and the formalisms used in our work (Section 2). Then follows with technical insights to implement STRAC (Section 3), research question formulation, experimental results with a discussion about them (Section 4). We then present related work (Section 5) and conclude (Section 6).

2 Background

In this section we discuss the key insights behind the non-determinism of the V8 bytecode generation process, as well as the foundations of the DTW alignment algorithm.

2.1 Browser Traces

Our dynamic analysis technique is evaluated with V8 bytecode [19]. In this subsection, we describe how the V8 engine generates bytecode trace. We collect such traces to evaluate our trace comparison tool. In this work, we use the term "V8 bytecode trace" to refer to the result of executing V8 with the *–print-bytecode* flag [21].

2.1.1 V8 Bytecode Generation

The V8 engine compiles JavaScript source code to an intermediate representation called "V8 bytecode". This is done to increase execution performance. The V8 engine parses and compiles every JavaScript code declaration present in HTML pages into a bytecode representation, composed by function declarations, like the one shown in Figure 1. These function declarations came from V8 builtin JavaScript code and external JavaScripts.

V8's bytecode interpreter is a register machine [16]. Figure 1 shows a JavaScript code and its bytecode translation.

Each bytecode operator specifies its inputs and outputs as register operands. V8 has 180 different bytecode operators.

The bytecode translation is lazy, i.e. V8 tries to avoid generating code it "thinks" might not be executed. Consequently, a function that is not called will not be compiled [28]. For example, removing line 2 in the top listing of Figure 1 would prevent the compilation of bytecode for the function declared in line 1. This behavior has an impact on the collected traces.

```
1  function plusOne(a){ return a.value + 1; }
2  plusOne( {value : 2018} );
```

```
1  [generated bytecode for function: plusOne]
2  Parameter count 2
3  Register count 0
4  Frame size 0
5  30 E> 0x1373c709b6 @ 0 : a5 00 00 00 StackCheck
6  56 S> 0x1373c709b7 @ 1 : 28 02 00 01
       ↪ LdaNamedProperty a0, [0], [1]
7  62 E> 0x1373c709bb @ 5 : 40 01 00 00 AddSmi [1],
       ↪ [0]
8  66 S> 0x1373c709be @ 8 : a9 00 00 00 Return
```

Figure 1. Example of a JavaScript function and its corresponding V8 bytecode instructions.

We have observed that V8 bytecode is resilient to script minification and static code-obfuscation techniques. Therefore, we believe that aligning such low-level representations could prove to be a useful aid in many program analysis tasks, such as code similarity study and malware analysis.

2.1.2 Non-Determinism in Browser Traces

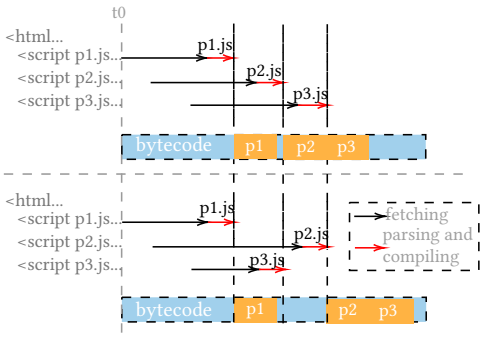


Figure 2. Illustration of two different script fetching and compiling traces for the same browser query.

Interestingly, browsers are fundamentally non deterministic, depending on web server availability, current workload,

and DNS caches through the network. Let us look at the example illustrated in Figure 2. It shows what happens when fetching a web page, which contains 3 scripts. The top and bottom parts illustrate, for the same page, two different executions. Dashed border rectangles represent complete bytecode generation traces. The blue spaces in the bar are V8 common builtin bytecode, which is systematically generated in all browser requests. Orange rectangles illustrate declared page scripts compilations. The complete bytecode trace is the union of both generated bytecodes, builtin V8 and page declared scripts. In the first case at the top of Figure 2, the scripts are fetched and compiled in the same order they are declared. In the second case, at the bottom, **p3.js** is carried and compiled first, before **p2.js** due to a possible network delay. However, V8's compiler will put all scripts compilations in the same order they are declared in the HTML page. The final result is two semantically equivalent bytecode compilations, where script blocks may not be strictly placed in the same position.

The slight differences that occur in the final bytecode for same browser queries motivate us to provide an efficient tool for traces alignment: traces where events occur in different orders but that have the same semantics must be considered as equivalent. The order of events should not confuse the trace comparison tool.

2.1.3 DTW Algorithm

The DTW algorithm has been introduced by Needleman and Wunsch for protein global alignment [18]. Global alignment means trace heads and tails are constrained to match each other in position. DTW is a popular technique for comparing traces in different domains, incl. software traces [14]. DTW finds the best global alignment between two traces, based on a generic similarity function between trace events and gaps.

Definition (Trace) A trace X is defined as a sequence of events. $X = x_1, x_2, ...x_N$ represents a trace of size N where each x_i is the event happening at the ith position.

Definition (Cost Matrix) D is a cost matrix for two traces X and Y of size n and m. D_{ij} stores the optimal cost alignment value for X and Y considered from the start up to the ith and jth positions respectively, that is the minimal cost of aligning x_i and y_j events at the same position in the final alignment.

The cost matrix is defined according to a distance function d and a gap cost γ as follows:

$$D_{0i} = \gamma * i$$

$$D_{j0} = \gamma * j$$

$$D_{ij} = min \begin{cases} D_{i-1j} + \gamma, \\ D_{ij-1} + \gamma \\ D_{i-1j-1} + d(x_i, y_j) \end{cases}$$

In every cell, the value D_{ij} is the minimum cost between putting a gap in one trace and the result of evaluating the distance function between events x_i and y_j.

Definition (Alignment Cost) Given two traces X and Y with sizes N and M respectively, the alignment cost is the value stored in D_{NM}.

Definition (Alignment Difficulty) Given two traces X and Y with sizes N and M respectively, the alignment difficulty is simply the multiplication of both sizes $N \times M$.

Definition (Warp Path) The warp path is the path to go from D_{NM} to the first element D_{00} minimizing the cumulative cost. In general more than one path may exist. Size of warp path is $O(N + M)$.

Definition (Aligned Trace) An aligned trace is a trace where the warp path is applied, i.e. some gaps have been put between some events in one of both traces.

In Figure 3 we illustrate the alignment between traces **abcababc** and **aabaca** with $\gamma = 1$, $d(x_i, y_j) = 2$ if $x_i \neq y_i$ and $d(x_i, y_j) = 0$ if $x_i = y_i$. The warp path is represented as the blue and orange lines going across the matrix from the top left corner to the bottom right corner. In this example, alignment cost is 4, as we can see in bottom right corner cell in Figure 3.

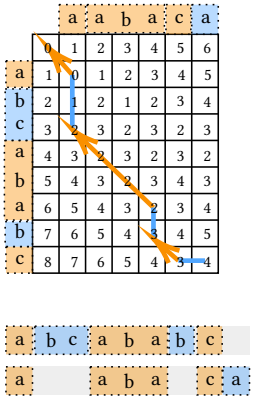


Figure 3. Cost matrix, warp path and applied alignment for **abcababc** and **aabaca** example traces.

3 STRAC: Trace Comparison Tool for V8

STRAC is an approach to compare large traces, tailored to bytecode traces of the V8 JavaScript engine. STRAC takes as input a trace of JavaScript V8 bytecode traces collected in the browser. It produces as output, a trace alignment, and a distance measure between the two traces. STRAC implements the DTW algorithm presented in Subsection 2.1.3. It is an open-source project publicly-available on https://github.com/KTH/STRAC . In this section, we explain the

key components and insights of STRAC to achieve scalable trace comparison.

3.1 Challenges Addressed by STRAC

Non-Determinism As shown in Subsection 2.1.2, V8 can provide two different bytecode traces for the same web page. In this case, both traces are semantically equivalent, but the global position of code modules can vary. These variations occur as a consequence of resource management, interpreter optimizations and JavaScript code fetching from the network. It is challenging because it can provide 1) false positives: two traces may be considered different even when they come from the same pages; 2) false negatives: two traces may be considered the same even when they come from two different pages.

Size Browser traces are huge and naive trace comparison fails on such traces because of memory requirements. For instance, aligning two traces of size 63137 and 58265 events requires a DTW cost matrix, represented as a bidimensional integer matrix, of 14.72 GB of memory. The challenge is to make trace comparison at the scale of browser traces, with tractable memory requirements.

3.2 DTW Distance Functions

The DTW algorithm has two main parameters: a distance function and a gap cost as explained in Subsection 2.1.3. The distance function between events affects the global alignment result, as we show in Subsection 4.5. It defines the matching of two different trace instructions if these instructions have a certain level of similarity. For example, when comparing *'AddSmi [0], [1]'* and *'AddSmi [1], [0]'* instructions, they can be considered as similar because the *AddSmi* operator is in both.

In STRAC, we define two distance functions for bytecode instructions.

$$d_{Sen}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ events are exactly} \\ \text{the same bytecode instruction} \\ c \text{ otherwise} \end{cases}$$

$$d_{Inst}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ bytecode instructions} \\ \text{share the same bytecode operator} \\ c \text{ otherwise} \end{cases}$$

Both require the identity relationship of the bytecode instruction. For V8 bytecode, based on our results (Subsection 4.5), it seems incoherent to accept an alignment match with two different elements instead of introducing the gap.

We now discuss the value of γ, s and c. The cost of introducing a gap, intuitively, must be less than the cost of matching two different events, i.e. $\gamma < s$. c is the value of matching two equal events, 0. The default values are based on our experience, $s = 5$, $\gamma = 1$ and $c = 0$. The three are configurable.

3.3 Buffering the Cost Matrix

The key limitation of DTW is the need for a large cost matrix to retrieve the warp path. Recall our example requiring 14.72 GB in Subsection 3.1. This means that a naive implementation can only compare small traces due to memory explosion.

In STRAC, we solve this problem by storing the cost matrix both in memory and disk. Only the appropriate values are kept in memory. Our key insight is that the current value D_{ij} in the cost matrix is calculated with the previous row and column, consequently, only $O(N)$ memory space is needed to compute D_{NM}. Thus, STRAC only maintains the current and previous row in memory for each DTW iteration. After processing a row, it is saved to disk. STRAC eventually saves the complete cost matrix to disk.

For traces with lengths 63137 and 58265, instead of 14.72 GB, STRAC requires no more than 86MB of memory for the trace alignment, which represents an improvement of 99.5% in memory consumption.

3.4 Retrieving the Warp Path

In addition to the alignment cost, it is necessary to obtain the warp path in order to create and analyze the aligned traces. Recall that the aligned traces are obtained by applying the warp path on both initial traces, as we mentioned in Subsection 2.1.3.

To retrieve the warp path from the final cost matrix, one goes backward and starts from the trace tail positions (D_{NM}). Cost matrix in D_{ij} depends on three neighbors D_{i-1j}, D_{ij-1} and D_{i-1j-1}. The backtracking process finishes when the trace start is reached, i.e. when the left top corner D_{00} is reached in the matrix. In the warp path construction process, trace indices are always decreasing by one, i.e. trace events are visited only once. Therefore, in STRAC, backtracking over the final cost matrix requires only $O(N + M)$ read operations on disk, which is scalable.

3.5 DTW Approximations

Due to the quadratic time and space complexity of DTW, previous work has proposed approximations to speed up the alignment process. STRAC also implements two state-of-the-art DTW approximations. We now mention these two approximations.

Fixed Regions Using fixed regions is a technique only to evaluate a specified region in the cost matrix [7, 12, 13, 24]. Consequently, the globally optimal warp path will not be found if it is not entirely in the window. This improvement speeds up DTW by a constant factor, but the execution time is still O(NM). STRAC provides support for fixed regions.

FastDTW [2] [25] is an approximation of DTW that has a linear time and space complexity. It combines data abstraction and constraint search in the solution space. STRAC implements FastDTW. Note that, for DTW and its approximations, the default mode is the buffering mode presented in Subsection 3.3.

3.6 Recapitulation

To sum up, STRAC is an optimized implementation of DTW and two approximations with distance functions dedicated to V8 bytecode traces and with neat handling of the cost matrix over memory and disk in order to scale.

4 Experimental Evaluation

We assess the scalability of STRAC for V8 bytecode trace comparison with the following research questions:

- RQ1 (Scalability): To what extent does STRAC scale to traces of real-world web pages?
- RQ2 (Consistency): To what extent does STRAC identify similarity in semantically-equivalent traces?
- RQ3 (Distance Functions): What is the effectiveness of STRAC support of different distance functions?

4.1 Study Subjects

Our experiment is based on tracing the home page of the following sites; google.com, github.com, wikipedia.org, youtube.com, four of the most visited websites, according to Alexa. We also add two sites based on personal interest: 2019.splashcon.org and kth.se, the homepage of our University. All those pages use JavaScript code. The traces were generated just opening the page without any other further action. Since the traces are non-deterministic, we collect 100 traces for the same page. This means we collect 600 traces in total.

Table 1. Descriptive statistics of our benchmark. The 6 sites are sorted by popularity according to the Alexa index. Example bytecodes are available in https://github.com/KTH/STRAC/tree/master/STRACAlign/src/test/resources/bytecodes.

Site	No. scripts	Bytecode size
google.com	5	85768
youtube.com	15	166626
wikipedia.org	4	48260
github.com	3	59384
kth.se	9	64178
2019.splashcon.org	17	147196

Table 1 gives an overview of the collected traces. The first column shows the real world website names. The second and third columns indicate the number of declared scripts and the bytecode size mean value (orange dots in Figure 4) respectively. For instance, Wikipedia loads 4 scripts and produces bytecode traces of 48260 bytecode instructions. This value is the lowest of our benchmark. On the contrary, for Youtube, the page declares 15 JavaScript scripts, and V8 generates traces of 166626 bytecode instructions, and this is due to the richer features of Youtube compared to Wikipedia. In our benchmark, the bytecode traces are in the range of 48k-166k instructions.

Recall that the bytecode traces are non-deterministic even for the same page (see Subsection 2.1.2). We measure how many instructions are contained in each V8 bytecode trace. Figure 4 illustrates the distribution of trace sizes as violin plots. This figure shows that there is a variance of bytecode traces for all pages (Wikipedia also has some variance but this is not shown in the figure because of the scale). This variance is a consequence of several stacked factors: resource management, interpreter optimization and JavaScript code fetching from the network. To our knowledge, this non-determinism in web traces is overlooked by research.

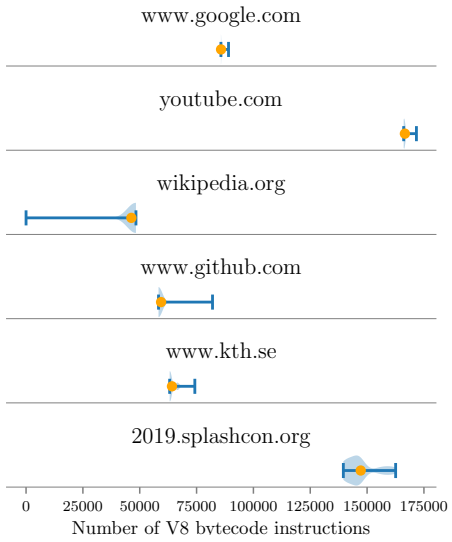


Figure 4. Variance of V8 bytecode trace size for 100 repetitions of the same query.

4.2 Experimental Methodology

Every trace is collected using a non-cached browser session, without plugins. This choice is motivated by two main reasons: 1) we have observed that cached scripts do not affect bytecode generation as direct network fetching does;

[2] The implementation mentioned in the original paper (https://cs.fit.edu/~pkc/FastDTW/) was not available at the moment of this work.

2) browser plugins are compiled to the same bytecode trace and in the scope of this work we are interested only in V8 bytecode traces directly generated from web page scripts.

To answer **RQ1**, we align 12 trace pairs randomly taken from the initial set of all possible trace pairs (600×600). We compare STRAC with different implementations of DTW 1) From public github repositories: rmaestre [3], dtaidistance [4] and pierre-rouanet [5]; 2) From R's dtw package [6] ; 3) The DTW implementation used in [15], slaypni [6]. For each comparison, we compute the average wall-clock execution time.

RQ2 is answered as follows. We select a random sample of 100 pairs from all possible trace pairs (600×600). We select 35 pairs of traces extracted from the same pages and 65 pairs of traces extracted from different pages. Alignment cost is measured for each pair using gap cost $\gamma = 1$ and event distance function d_{Sen} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We group and plot each pair alignment cost per site.

We answer **RQ3** using the same traces as *RQ2*. We compute DTW on each one of the 100 sampled pairs. We use the same gap cost $\gamma = 1$, but we compare the two distance functions d_{Sen} and d_{Inst} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We measure the alignment cost for each pair and compare the results with the ones obtained in *RQ2*.

The STRAC experimentation has been made on a PC with Intel Core i7 CPU and 16Gb DDR3 of RAM. We extract all traces from Chrome version 74.0.3729.169 (Official Build) (64-bit).

4.3 Answer to RQ1: Scalability

Figure 5 shows the execution time of 6 different alignment tools on 12 trace pairs. The X axis gives the size of the alignment problem, which is the multiplication of the size of both traces in number of bytecode instructions. The Y axis represents the execution time in seconds with a logarithmic scale.

First, we observe that four tools get out of memory for all the considered trace pairs: R-dtw, cpy-wannesm, rmaestre, cpy-slaypul (see the red dot in Figure 5). The main reason for this failure is that those tools need to store the cost matrix in memory. The least difficult trace comparison in the plot is a pair of traces of 48k instructions each. Finding the best alignment for this pair consists in analyzing an eight-bytes integer matrix of approx. 20GB (exactly 18632 millions of bytes). This memory requirement is almost the full memory of modern personal computers and it causes memory explosion at runtime. Applying the same analysis to the most difficult alignment in the plot shows requires 200GB of memory.

[3] https://github.com/rmaestre/FastDTW
[4] https://github.com/wannesm/dtaidistance
[5] https://github.com/pierre-rouanet/dtw
[6] https://github.com/slaypni/fastdtw

Second, py-wannesm and py-pierre-rouanet calculate the best alignment cost for the first 10 pairs, without any memory issue, even for problems in the order of magnitude close to 1.5×10^{10} in alignment difficulty. After this value, these tools also start to get memory issues for the same reason as the other tools. Yet, these succesfully align the 10 pairs (orange and green curves in Figure 5) thanks to an efficient use of Numpy [3] arrays to store cost matrix. Numpy arrays in Python are tailored to efficiently deal with arrays up to 20GB of memory in x64 architectures. We also observe that py-wannesm is always slower than py-pierre-rouanet. The main reason for this time difference is that py-wannesm does an extra pass through the cost matrix and py-pierre-rouanet does not do it.

Third, STRAC succesfully find the best alignment cost for all pairs in the benchmark, even for trace pairs that require memory beyond Numpy capabilities (the last two blue dots in Figure 5). The key insight behind is that STRAC implements the cost matrix data structure as a hybrid between memory and disk, i.e. moving such memory needs to disk.

Both Python implementations (py-wannesm and py-pierre-rouanet) systematically take at least one order of magnitude longer to run, compared to STRAC. The main reason behind this is that Python usually compiles code at runtime, while Java compiles it in advance, making a faster program. Besides, most JVMs perform Just-In-Time compilation to all or part of programs to native code, which significantly improves performance, but mainstream Python does not do this.

Recall that best alignment calculation using naive DTW implementation is non-scalable by its space-time quadratic nature, any implementation of DTW (even the one included in STRAC) eventually will run out of space (in memory or disk) and execution time will be near to impossible. However, STRAC can deal with all trace pairs of our benchmark thanks to its hybrid strategy that leverages both the disk and the memory. To align an average trace of 100k instructions, STRAC takes approx. 14 minutes in a PC like the one mentioned in Subsection 4.2.

4.4 Answer to RQ2: Consistency

In Figure 6, we plot the alignment cost for 100 trace pairs, the blue dots represent pairs extracted from the same page, the orange dots illustrate trace pairs taken from two different pages. Each column corresponds to a given web page. Green dots represent pairs with the maximum alignment cost for each site: an alignment of the web page treated in the column with a trace from the site cited above the dot. For example, the green dot in the first column is an alignment of a trace pair (2019.splashcon, youtube).

In Figure 6, we observe that, for each site, traces from the same page have a lower alignment cost. This is consistent with the fact that in these cases, the majority of both traces

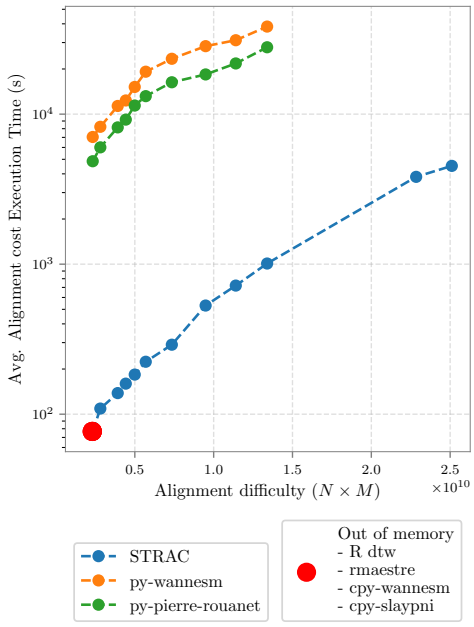


Figure 5. Execution time for **12** trace pair comparisons by 7 tools incl. STRAC. Y axis is in logarithmic scale. Four tools fail even on the smallest traces.

in the pair are the same. On the contrary, the alignment cost between traces from different pages is higher.

Some cases show blue dots with sparsed high values. This occurs when external scripts, declared in some pages, present a high variance in fetching process time. Also, it sometimes happens that for one script declared in a page, the remote servers sends different JavaScript code at each every request. Therefore, the generated bytecode varies more from one load to another, and the alignment cost is increased, showing a small margin between orange dots and the blue ones. However, we observe two scenarios when these phenomena are mitigated. First, when the bytecode generated from the external declaration is larger than the builtin bytecode (2019.splashcon, UNIV, and Youtube cases present a clear separation between clusters). Second, when the fetching process time is stable, as Wikipedia and Github cases show.

In the case of Google, we observe the worst possible scenario. This site has 5 external declared scripts (see Table 1), 3 of them have variable fetching time and their content varies at each load. These 3 scripts integrate Google Analytics features to the site. On the contrary, in the case of Wikipedia, external declared JavaScripts always provide the same code in almost constant time. As a result, the generated bytecode is more deterministic and alignment cost decreases for traces from the same site. In the case of Wikipedia, alignment costs for pairs of traces collected from the same page vary between 1926 and 2652. These values are the lowest alignment costs in the benchmark, and they differ from others in more than 2× in order of magnitude

Overall, the traces from the same (resp. different) page are located in separated clusters. In all cases, we also observe groups of orange dots that can be easily separated from other orange clusters. This separation is a consequence of semantic differences between sites and the increase of JavaScript declarations. For instance, in the first column of Figure 6, trace pairs from 2019.splashcon and Youtube home pages have higher alignment costs. This is a consequence of that Youtube is a richer feature site as 2019.splashcon is, but they semantically differ. We also observe this behavior in the case of Kth and Youtube trace pairs.

V8 compiles builtin JavaScript code to the same bytecode trace, as we discussed in Subsection 2.1.1. This bytecode generation is included in all collected traces. To validate this, we computed the V8 bytecode trace of an empty page: it contains 40k bytecode instructions on average. This also represents a constant noise in the alignment computation.

As Figure 6 illustrates, given the alignment cost of two semantically equivalent traces (blue dots) as a reference, STRAC is capable of identifying similarity with other page traces. However, we want to remark that STRAC accuracy gets improved when JavaScript declarations increase in the compared sites.

4.5 Answer to RQ3: Distance Functions

In Figure 7, we plot the alignment cost using distance d_{Ins}. Recall that d_{Ins} is less restrictive than d_{Sen}, the distance used to answer **RQ2**. By comparing Figure 7 and Figure 6, we observe interesting phenomena. First, changing the distance function breaks the clustering breakdown for Github, Google and Kth (some blue points get mixed with orange points). Second, the maximum alignment cost is lower than in Figure 6 for all sites. These phenomena are consequences of using a less restrictive distance function, i.e. with d_{Ins}, only the operator is analyzed in the bytecode instructions comparison. Overall, the choice of distance function matters. STRAC can be extended with new distance functions and provides d_{Sen} by default for properly aligning V8 bytecode traces.

We notice that the impact of the distance function is bigger for sites with less JavaScript. For Google, Github and Wikipedia, using d_{Ins} is bad because it breaks the clustering. For the remaining three websites, which involve more JavaScript features, while the alignment changes, the core property of the alignment of identifying semantically equivalent traces still holds.

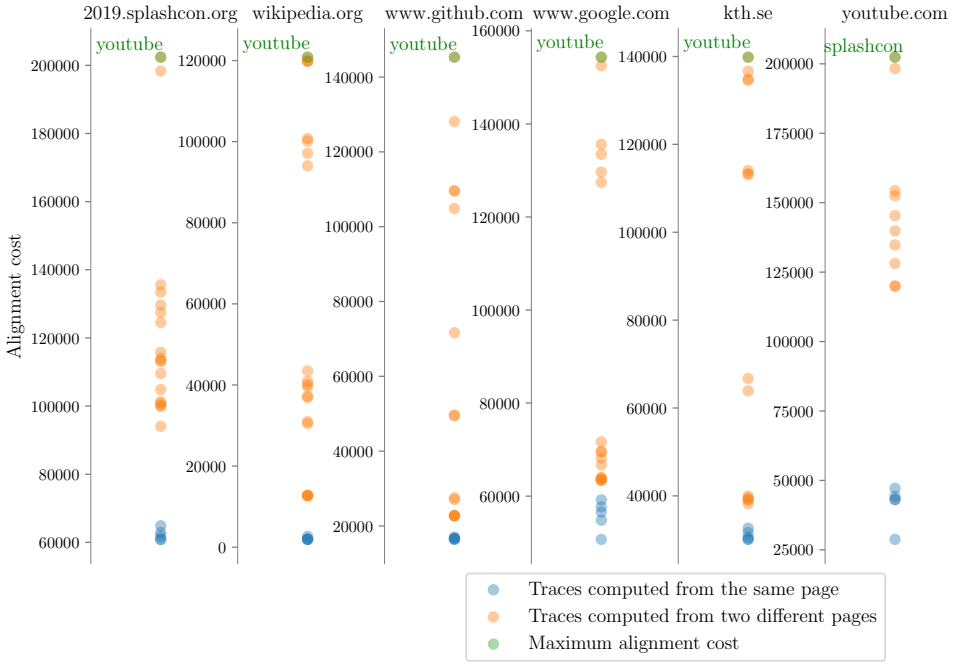


Figure 6. Alignment costs for 100 trace pair comparisons using d_{Sen} as distance function.

5 Related Work

DTW is memory greedy on trace size, a similar problem arises when dealing with streaming traces. Oregi et al. [20] and Martins et al. [15] present a generalization of DTW for large streaming data. They propose the use of incremental computation of the cost matrix complemented with a weighted event distance function adding event positions. However, their results may differ from the original DTW warp path. On the contrary, STRAC also computes the exact alignment cost without approximations.

Kargen et al. [10] propose a combination of data abstraction and FastDTW to align two program traces at the binary level. They record and analyze read and write operations to memory and x86 registers. Also, they argue and they show that their method scales to large traces. STRAC is also capable of analyzing such traces, but targets different kinds of traces: V8 bytecode traces, which are not handled by Kargen et al.

Ratanaworabhan et al. [23] instrument Internet Explorer to measure JavaScript runtime and static behavior in function calls and event handlers on real-world websites. By doing so, they show that common benchmarks, like SpiderMonkey and V8-Suite, are not representative of real application behavior. We could use STRAC to perform a similar analysis on modern browsers.

With JALANGI, Sen et al. [26] provide a framework to dynamically analyze JavaScript. The framework works through source code instrumentation. JALANGI associates shadow values to variables and objects in the instrumented code, Sen et al. argue that most of of state-of-the-art dynamic analysis techniques can be implemented, like concolic evaluation and taint analysis. However, JALANGI has several limitations dealing with builtin code and instrumentation can decrease instrumented code execution performance. With STRAC, we propose to use V8 bytecode traces to compare JavaScript semantic similarity without JavaScript instrumentation.

Fang et al. [5] propose a JavaScript malicious code detection model based on neural networks. To mitigate the obfuscation techniques used in malicious code, they analyze the dynamic information recorded in V8 bytecode traces. Both STRAC and Fang et al. consider V8 bytecode traces, yet the usages are different: they do anomaly detection while we do trace comparison.

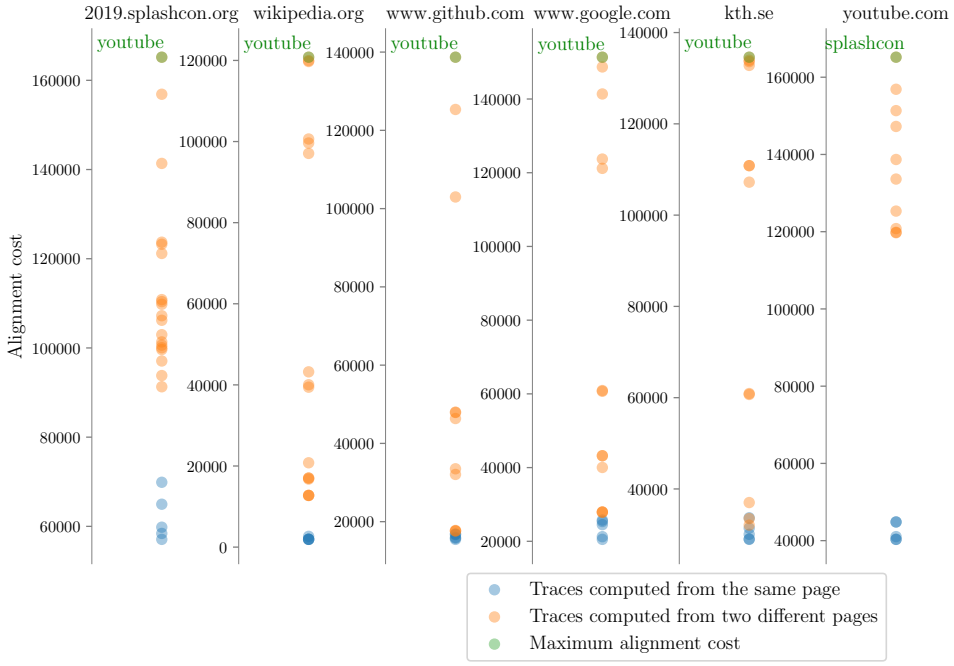


Figure 7. Alignment cost for 100 trace pair comparisons using d_{Ins} as distance function.

6 Conclusion

In this paper, we presented a tool, called STRAC, for aligning execution traces. STRAC is tailored to traces of the JavaScript V8 engine. STRAC implements an optimized version of the DTW algorithm and two of its approximations. Our experiments show that STRAC scales to real-world JavaScript traces consisting of V8 bytecodes. STRAC provides two distance functions for trace event comparison and can be configured with any arbitrary distance function. Our evaluation indicates that STRAC performs better than state of the art DTW implementations, for 6 representative web sites.

We have shown that V8 bytecode contains redundancy and that an empty page includes more than 40k trace instructions. By removing this redundant and useless trace instructions, the alignment would get better. In our future work, we will study how to remove redundancy in V8 bytecode traces, for providing a better behavioral similarity measure for modern web pages full of JavaScript code.

Acknowledgments

This material is based upon work supported by the Swedish Foundation for Strategic Research under the Trustfull project and by the Wallenberg Autonomous Systems and Software Program (WASP).

References

[1] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering - SIGSOFT/FSE '11* (2011). ACM Press, 267. https://doi.org/10.1145/2025113.2025151

[2] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

[3] Numpy community. 2018. Numeric python. https://www.numpy.org/index.html

[4] V8 JavaScript engine. 2016. *Ignition design documentation.* https://v8.dev/docs/ignition

[5] Y. Fang, C. Huang, L. Liu, and M. Xue. 2018. Research on Malicious JavaScript Detection Technology Based on LSTM. *IEEE Access* 6 (2018), 59118–59125. https://doi.org/10.1109/ACCESS.2018.2874098

[6] Toni Giorgino. 2009. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package. *Journal of Statistical Software, Articles* 31, 7 (2009), 1–24. https://doi.org/10.18637/jss.v031.i07

[7] F. Itakura. 1975. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23, 1 (February 1975), 67–72. https://doi.org/10.1109/TASSP.1975.1162641

[8] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. 2007. Multiresolution Abnormal Trace Detection Using Varied-Length *n*-Grams and Automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37, 1 (Jan 2007), 86–97. https://doi.org/10.1109/TSMCC.2006.871569

[9] T. Kamiya. 2018. Code difference visualization by a call tree. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 60–63. https://doi.org/10.1109/IWSC.2018.8327321

[10] Ulf Kargén and Nahid Shahmehri. 2017. Towards Robust Instruction-level Trace Alignment of Binary Code. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=3155562.3155608

[11] Hyunjoo Kim, Jonghyun Kim, Youngsoo Kim, Ikkyun Kim, Kuinam J. Kim, and Hyuncheol Kim. 2017. Improvement of malware detection and classification using API call sequence alignment and visualization. *Cluster Computing* (12 Sep 2017). https://doi.org/10.1007/s10586-017-1110-2

[12] Daniel Lemire. 2008. Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound. *CoRR* abs/0811.3301 (2008). arXiv:0811.3301 http://arxiv.org/abs/0811.3301

[13] Y. Lou, H. Ao, and Y. Dong. 2015. Improvement of Dynamic Time Warping (DTW) Algorithm. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 384–387. https://doi.org/10.1109/DCABES.2015.103

[14] Marcelo De A. Maia, Victor Sobreira, Klérisson R. Paixão, Ra A. De Amo, and Ilmério R. Silva. 2008. Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*. 6–10.

[15] R. M. Martins and A. Kerren. 2018. Efficient Dynamic Time Warping for Big Data Streams. In *2018 IEEE International Conference on Big Data (Big Data)*. 2924–2929. https://doi.org/10.1109/BigData.2018.8621878

[16] Ross McIlroy. 2016. Ignition: V8 Interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit

[17] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. https://doi.org/10.1109/ICSME.2014.37

[18] Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[19] V8 official web page. 2019. *V8 JavaScript Engine*. https://v8.dev/

[20] Izaskun Oregi, Aritz Pérez, Javier Del Ser, and José A. Lozano. 2017. On-Line Dynamic Time Warping for Streaming Time Series. In *Machine Learning and Knowledge Discovery in Databases*, Michelangelo Ceci, Jaakko Hollmén, Ljupco Todorovski, and Saso Vens, Celinand Dzeroski (Eds.). Springer International Publishing, Cham, 591–605.

[21] The Chromium Projects. 2019. *Run Chromium with Flags - The Chromium Projects*. https://www.chromium.org/developers/how-tos/run-chromium-with-flags#TOC-V8-Flags

[22] David A Ramos and Dawson R. Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 669–685. http://dl.acm.org/citation.cfm?id=2032305.2032360

[23] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1863166.1863169

[24] H. Sakoe and S. Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 1 (February 1978), 43–49. https://doi.org/10.1109/TASSP.1978.1163055

[25] Stan Salvador and Philip Chan. 2007. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.* 11, 5 (Oct. 2007), 561–580. http://dl.acm.org/citation.cfm?id=1367985.1367993

[26] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[27] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. TraceDiff: Debugging Unexpected Code Behavior Using Trace Divergences. *CoRR* abs/1708.03786 (2017). arXiv:1708.03786 http://arxiv.org/abs/1708.03786

[28] Toon Verwaest and Marja Hölttä. 2019. *Blazingly Fast Parsing, Part 2: Lazy Parsing · V8*. https://v8.dev/blog/preparser

[29] M. Weber, R. Brendel, and H. Brunst. 2012. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. 247–254. https://doi.org/10.1109/ISPA.2012.40